FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Rust blockchain networks
Student:	Bc. Tomáš Rokos
Supervisor:	Ing. Josef Gattermayer, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2023/2024

Instructions

Fast and low-cost blockchain technology is crucial for the widespread adoption of blockchain in various industries. High transaction speed and low costs are necessary for blockchain to be able to handle a large volume of transactions, such as in the case of financial institutions and supply chain management. The goal of this thesis is to compare new blockchain technologies and possibly contribute to their toolings for better development experience.

- Provide background information on the importance and potential of blockchain technology, as well as the current challenges facing the widespread adoption of blockchain, such as cost and scalability.
- Summarise the current state of research on fast and low-cost blockchain technologies, including but not limited to technologies such as Solana, Aptos, Cosmwasm and others. Compare them to each other and to currently widely adopted technologies like Ethereum in terms of their architecture, state of developer and security tooling, and other relevant aspects.
- Select one of the blockchain technologies that were compared in the previous section, and write a contribution to an existing security/development tool for that network, or create a new security/development tool if none exists. Possible areas are implementation or improvement of testing framework, fuzzy tests, VS Code extension, debugger or another blockchain based development tooling.
- Test the tool, compare it to existing tooling on other chains (preferably Solana).

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 23 January 2023 in Prague.

Master's thesis

RUST BLOCKCHAIN NETWORKS

Bc. Tomáš Rokos

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Josef Gattermayer Ph.D.
May 4, 2023

Citation of this thesis: Rokos Tomáš. *Rust blockchain networks.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I want to thank the supervisor, Ing. Josef Gattermayer, Ph.D., for his guidance and swift communication throughout the whole process of writing this thesis. Additionally, my appreciation goes to Ing. Adam Hrazdira for his valuable support in leveraging the advanced functionalities of the Anchor framework. I would like to thank my family for their support and understanding.

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 May 4, 2023 ....

Abstract

This thesis compared Rust blockchain networks in terms of their architecture, developer convenience, test, and security tooling. Blockchain technology Solana was chosen for improvement, and Trdelnik, a testing framework, was enhanced with parallelized runtime and fuzz testing capabilities. The improved framework showed significantly faster processing times, and the fuzz tester was evaluated on multiple smart contracts.

Keywords blockchain, testing framework, fuzz tests, rust, solana

Abstrakt

Tato diplomová práce porovnává blockchainové sítě napsané v jazyce Rust z hlediska architektury, přívětivosti vývoje a stavu testovacích a bezpečnostním nástrojů. Blockchainová technologie Solana byla vybrána k vylepšení testovacího frameworku Trdelník. Framework byl rozšířen o možnost paralelního běhu testů a využívání fuzz testů. Vylepšená verze Trdelníku zrychlila běh testů a fuzz testy byly použity k testování několika tzv. smart contractů.

Klíčová slova blockchain, testovací framework, fuzz testy, rust, solana

List of abbreviations

ABI	Application Binary Interface. 6, 9, 12, 34
API	Application Programming Interface. 7, 50, 69
BPF	Berkley Packet Filter. 33, 70
CLI	Command Line Interface. 65, 69
DApp	Decentralized Application. 7, 32
DPoS	Delegated Proof of Stake. 21
EVM	Ethereum Virtual Machine. 6
IDE	Integrated Development Environment. 7, 18, 31, 41
IDL	Interface Definition Language. 34
IoT	Internet of Things. 1
LLVM	Low Level Virtual Machine. 8
MSL	Move Specification Language. 11
PDA	Program Derived Address. 34, 70
PoA	Proof of Authority. 29
PoH	Proof of History. 33
PoS	Proof of Stake. 6, 21, 29, 33
PoW	Proof of Work. 6
RPC	Remote Procedure Call. 7
SSO	Single Sign On. 29
TDD	Test Driven Development. 11, 16
VSCode	Visual Studio Code. 18, 23, 31, 37, 40
WASM	Web Assembly. 29, 30, 41
XSS	Cross-site scripting. 29

Introduction

Blockchain technology has influenced the way we develop software and publish data in many industries, including IoT [1], health care [2], and finance [3]. Scaling this technology to the size of current internet services in these industries has become an emerging problem [4]. While VISA claim they can handle up to 24 000 transactions per second [5], the cryptocurrencies with the highest market capital, like Bitcoin and Ethereum, can only handle around 7 and 40 transactions per second, respectively [6].

When blockchain is scaled to higher transaction rates, tradeoffs must be made between scalability, security, and decentralization, as Vitalik Buterin, the founder of Ethereum, stated in the blockchain trilemma [4].

Public instructions running in a decentralized environment means there is a more significant attack vector for hackers, as there is no way of changing the blockchain state other than forking the entire network. When it comes to security, blockchain technology is still in its infancy. No Ethereum test tools have been able to detect all 13 widely known vulnerabilities [7]. The pressure on developers to write secure code when developing smart contracts is high, as the consequences of a bug can cost millions of dollars. Incidents like the Parity Wallet and the DAO hack led to a loss of 150 million dollars [8].

Tools for developing programs running on blockchain nodes are needed to achieve the best tradeoff between scalability, security, and decentralization. They should guide developers to write secure programs. The blockchain network ecosystem should provide the necessary tools for testing and debugging.

The first chapter will cover the aim of this thesis and what should be the output coming from the research. The second Technology chapter will cover an overview of used technologies and why they were chosen. The third chapter will review the methodology of researching individual blockchain networks. The analysis chapter will then cover the results of the research. Practical chapter five will go over the reasons for the picked technology and the development of the security tool. The tool will be evaluated in the last chapter.

Chapter 1

Aim of thesis

The theoretical part should provide background information on the current challenges facing the widespread of blockchain, such as cost and scalability. It should include a comparison of fast and low-cost blockchain technologies such as Solana, Aptos, Sui, and Near with themselves and widely adopted technologies such as Ethereum. The comparison will be made regarding the developer convenience, testing framework, static analysis tool, IDE Support, and wallet and software integration. There will also be an overview of the architecture of blockchain technology from its whitepaper.

Upon completing the theoretical part, the practical part will focus on the chosen blockchain technology based on the retrieved information. The aim is to improve or create a new security/development tool to be used in the development of smart contracts.

The tool should be evaluated on multiple examples and compared to other existing tools for blockchain technologies. The evaluation should include ease of use and the tool's performance.

Chapter 2
Technology

The rapid advancement of technology has given rise to many innovations, including the emergence of blockchain technology. This chapter provides an overview of blockchain technology, ways of achieving consensus between decentralized participants on the network, and cryptographic keys.

Specific technologies like Rust and Move programming languages will also be explained in the context of writing smart contracts. The constructs of the individual languages that could improve safety will be mentioned.

2.1 Blockchain

A blockchain is a form of database with specific properties that make it unique. A unit of data will be called a transaction. It is immutable, meaning it cannot be changed once a transaction is added to the blockchain. It is cryptographically verifiable that the order of data in the blockchain is correct. It is also verifiable that the data in the past has not been changed. It is also decentralized, meaning that there is no central authority that controls this database.

Blockchain is also shared between all network participants, meaning multiple database copies exist. There needs to be some way of achieving consensus among all participants in the network when appending new data to the database. The consensus is done by using a consensus algorithm. The algorithm decides what would be the subsequent transactions included in the blockchain.

Transactions as atomic units of data are grouped in blocks. A block is a collection of transactions that are validated by the consensus algorithm.

2.1.1 Blockchain nodes

Not all participants in the network want to store the entire copy of the database. The database can be extensive. All participants do not need to store the entire database. There are different types of participants in the network. In Bitcoin, there are three types of nodes. Full nodes store the entire database. Mining nodes validate transactions and submit new blocks to the blockchain. Then there are light nodes, which only store the headers of the blocks. If a participant in the network wants to send transactions, he will act just as a light node. [9]

When light nodes add transactions or data onto the blockchain, they would have to trust the full nodes not to tamper with the data. The full node must also validate the sent transaction. Node decides whether the sender can append this transaction onto the blockchain. The problem with authentication is solved by generating a cryptographic keypair (public key and private key) using an encryption algorithm defined in the standard. Every participant can then sign the transaction with their private key. The full node can verify the signature with the sender's public key. Keypairs can be used in the cryptocurrency blockchain to validate if the sender has enough funds and owns them, as they are saved under his public key.

A basic example of a blockchain could be a linked list, and every node in that list has multiple transactions. Every block is hashed to make the blockchain immutable, and the previous block's hash is added to the current block. This way, the order of the blocks cannot be changed without changing the current block's hash. The consensus mechanism would decide which node can submit the following block to the linked list.

2.1.2 Consensus algorithms

Widely used blockchain consensus mechanisms are Proof of Work (PoW) and Proof of Stake (PoS).

Proof of Work is a consensus algorithm where the participants in the network compete to solve a mathematical problem. The first participant in solving the problem will be able to submit the following block to the blockchain and be rewarded. If the participant tries to submit an invalid block, every other participant in the network will reject the block, and he loses the reward.

Proof of Stake is a consensus algorithm where the participants in the network stake a certain amount of cryptocurrency. The participants are then chosen in some fashion. It could be, for example, randomly or based on the number of tokens staked. The participants that are chosen to submit the next block are rewarded. If the participant tries to submit a block that is not valid, he loses this stake.

2.1.3 Smart Contracts

Having a blockchain used only for financial transactions is not the only use case for blockchain technology. When sending a transaction using Bitcoin, the validation of the transaction itself is not embedded inside the blockchain source code. Instead, there is a stack-based scripting language called Bitcoin Script [9]. Bitcoin Script makes it possible to add logic to the transaction itself, like a transaction puzzle where the sender has to solve a puzzle to be able to complete the transaction or require two different private keys to sign the transaction. However, the language is not Turing complete, meaning that problems like infinite loops did not have to be addressed in the protocol's design. The validators would not be slowed down by the transactions' validation, which means the language cannot define any complex logic. [10]

Both Ethereum and Bitcoin came up with their own instruction set called Opcodes. The nodes validating transactions have a virtual machine running these instructions. [11], [12]

Ethereum's opcodes set is more extensive as it came up with a solution that makes it possible to embed actual programs using Turing complete language on the blockchain. The language is called Solidity, which compiles to Ethereum Virtual Machine (EVM) bytecode. The EVM is a stack-based virtual machine that interprets the EVM bytecode. The mining nodes in the network run this bytecode when validating the transaction making the blockchain network a decentralized computer with its state. Ethereum is, therefore, extensible beyond just financial transactions.

It solved the problem of having a Turing complete language on the blockchain with a gas limit for each transaction. The gas limit is the amount of computational power the transaction can use. The gas limit is also used to calculate the transaction fee. The transaction fee is paid to the miner that validates the transaction. Gas fees make it possible to have a Turing complete language running in a decentralized environment while ensuring that the transaction validation does not slow the network down. [13]

The code deployed in the transaction on the blockchain is called a smart contract in the Ethereum ecosystem. The code has an Application Binary Interface (ABI) which defines an interface of methods that can be used to interact with the contract. The ABI defines the input and output parameters of the methods. Network participants can interact with the contract and any other smart contract that can interact with the contract using the ABI.

Having a Turing complete language running on the blockchain comes with the problem of having a public code in a decentralized environment where anyone can read the code. When the code is deployed into the environment, malicious users can read the bytecode and try to exploit it. There is no rollback mechanism, and the blockchain is append-only, meaning that once the exploit is used maliciously, there is no way of recovering the funds. There is a possibility of upgrading the code of the contract in Ethereum. The users of the contract have to trust the contract's developers, which lowers the blockchain's trustless, decentralized nature.

Like normal software development with code pushing from the development environment through testing to production, blockchain technologies release networks allowing developers to go through similar processes.

There is devnet which serves for testing while developing the smart contract. Devnet could be just a local machine or online IDE used for testing while developing the code. Then there is the testnet which is used for stress testing the smart contract or demo purposes. Both of these networks allow the user to airdrop cryptocurrency tokens for free with some limitations. These networks do not have to be decentralized and could use consensus mechanisms that do not require as much computing power. Then there is mainnet which is the production environment.

2.1.4 Remote Procedure Calls

When interacting with the blockchain, the nodes can implement a Remote Procedure Call (RPC) API that can be called to interact with the blockchain. The RPC allows remotely calling methods predefined by the protocol with the option of specifying input and output types.

Ethereum uses the JSON-RPC, which implements the RPC protocol using JSON. It can be used over both sockets and HTTP.

2.1.5 Decentralized applications

Decentralized Application (DApp) are applications with some frontend that interacts with the blockchain as the backend. The frontend can be a website, a mobile application, or a desktop application that uses API defined by the protocol.

When wanting to change the blockchain's state, the nodes must sign the transaction to prove its validity. The frontend of the application needs to have some way of getting a signature from the user.

In website applications, the signing mechanism comes as a browser extension. The user must trust the browser extension as it can sign transactions on its behalf and read the user's private key. The software capable of signing the transactions and holding the private key is called a wallet.

2.2 Rust programming language

This diploma thesis specializes in blockchain technology which supports deploying smart contracts written in Rust programming language or is itself written in Rust. The intention of this section is not to provide a comprehensive overview of the Rust programming language but rather to highlight the features of the Rust programming language that make it suitable for developing smart contracts and cryptocurrencies.

”Rust is a new programming language for developing reliable and efficient systems. It is designed to support concurrency and parallelism in building applications and libraries that take full advantage of modern hardware. Rust's static type system is safe and expressive and provides strong guarantees about isolation, concurrency, and memory safety.

Rust also offers a clear performance model, making it easier to predict and reason about program efficiency. One important way it accomplishes this is by allowing fine-grained control over memory representations, with direct support for stack allocation and contiguous record storage. The language balances such controls with the absolute requirement for safety: Rust's type system and runtime guarantee the absence of data races, buffer overflows, stack overflows, and accesses to uninitialized or deallocated memory." [14]

The cited article is from Rust's authors and well captions Rust's main features. It was an internal project of Mozilla used for their rendering engine of the Firefox browser. While in 2014, the Rust language was a new programming language used primarily by Mozilla, nowadays it is a mature language used by many open source projects, including Linux kernel [15]. In 2021 a nonprofit organization called Rust Foundation, which focused on supporting Rust developers was created [16]. Therefore its future is not tied only to Mozilla anymore.

The unique combination of memory-safe statically typed compiled language without a garbage collector and the ability to write low-level code makes Rust a good candidate for writing smart contracts. It uses the Low Level Virtual Machine (LLVM) project as a frontend for the language, meaning it can be compiled to a variety of different instruction sets implemented for LLVM. That is beneficial as the compiled program needs to run in the specific sandboxed environment defined by blockchain technology.

Memory safety is achieved using the ownership system of Rust. The ownership system is a compile-time feature of Rust that allows Rust to free memory automatically.

2.2.1 Error handling

```rust
fn divide(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        return Err("The divisor can not be zero.".to_string());
    }
    Ok(a / b)
}

fn main() {
    let res = divide(5.0, 0.0);
    match res {
        Ok(val) => {
            println!("Value is {val}")
        },
        Err(e) => {
            println!("The function ended with error: {e}")
        }
    }
}
```

Code listing 1 Example of using a Result type

It uses an error-handling approach similar to that used in functional programming languages. It has no exceptions like language such as C++ or Java, as exceptions are essentially a goto inside a program. Error handling using the Result type is very similar to Either in functional programming. It can be either a `Ok(T)` or an `Err(E)` with specified types `Result<T, E>`. The developer then needs to handle the error in the code. Otherwise, the underlying `T` value can not be accessed. It is impossible to forget to handle the error. An example of such code is in 1.

It also has no concept of null value, which Sir Anthony Hoare, inventor of the Quicksort algorithm and Algol 60 compiler [17] called his Billion dollar mistake [18]. It instead uses the

Option type, which is an enum with two variants `Some(T)` and `None`. The type is very similar, requiring the programmer to check if a type is not null before accessing the value.

2.2.2 Macros

The language itself has first-class support for macros. This makes the language very customizable for cryptocurrency-specific use cases. The macro is called using the `#[macro_name]` syntax.

```rust
use near_sdk::borsh::{self, BorshDeserialize, BorshSerialize};
use near_sdk::near_bindgen;

#[near_bindgen]
#[derive(BorshDeserialize, BorshSerialize, Default)]
pub struct Counter {
    value: u64,
}

#[near_bindgen]
impl Counter {
    pub fn increment(&mut self) {
        self.value += 1;
    }

    pub fn get_count(&self) -> u64 {
        self.value
    }
}
```

Code listing 2 Near definition of a contract

Code listing 2 is an example from the Near documentation [19]. The `#[near_bindgen]` macro is used to generate the ABI for the contract automatically by just defining the methods and the state of the contract. It takes the default rust definition of public and private and generates the ABI accordingly.

There are macros for serialization and deserialization of the data inside of the struct as well. The data in example 2 is serialized using the `Borsh` serialization format. Data is therefore saved in binary format, which is more efficient for storage as the data is stored in possibly thousands of nodes across the network. The serialization and deserialization of the data are seamless and done automatically with the help of macros.

The code listing 2 also shows the mutability feature of Rust. The `&mut self` syntax defines a mutable reference to the struct. The struct can then be modified inside of the function. The `&self` syntax defines an immutable reference to the struct. The struct can not then be modified inside of the function. When trying to mutate a variable that is not declared mutable, the code would not compile.

2.2.3 Concurrency

Rust is also an excellent fit for developing the software for the nodes. Their documentation defines a concept called fearless concurrency. Because of the ownership feature and specific mutability, it can catch many issues regarding memory safety at compile time. For example, when accessing data from multiple threads, the code would not compile as race conditions would

```rust
use std::sync::Mutex;
use std::thread;
fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];
    for _ in 0..10 {
        let handle = thread::spawn(|| {
            // You can not mutate or read the value before acquiring the lock
            let mut num = counter.lock().unwrap();
            *num += 1;
            // The lock is automatically released
            // when the variable goes out of scope
        });
        handles.push(handle);
    }
    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Code listing 3 Using Mutex from The Rust Book [20]

be possible. The compiler requires to use of a structure like `Mutex<T>` that would ensure that only one thread can access the data at a time. In the same way, data could not be accessed from a `Result<T, E>` type before checking the error. Here the data could not be accessed from a `Mutex<T>` type before locking it. It also automatically unlocks the Mutex upon leaving the scope of the function, as seen from the code example 3. [20]

When developing a decentralized, trustless protocol, nodes must synchronize their state. Ethereum has a gossip protocol that handles the propagation of transactions and blocks between nodes. The authors in On block delivery time in Ethereum network [21] claim that a new block arrives every 12 to 14 seconds. When dealing with networking and synchronization, the asynchronous programming technique is excellent, as the node does not have to block threads until it receives a response. Rust has first-class support for asynchronous programming, and there is a possibility of using async await syntax to write asynchronous code that looks synchronous.

2.3 Move programming language

”Move is a programming language for writing safe smart contracts originally developed at Facebook to power the Diem blockchain. Move is designed to be a platform-agnostic language to enable common libraries, tooling, and developer communities across various blockchains with vastly different data and execution models. Move’s ambition is to become the ”JavaScript of web3” in terms of ubiquity–when developers want to quickly write safe code involving assets, it should be written in Move.” [22]

Even with the extensibility of Rust using macros, it was not a language designed for building smart contracts. Having a domain-specific language for just smart contracts means that it is possible to add valuable features only when developing them.

2.3.1 Platform agnostic

The problem with domain-specific languages is that they require developers to learn the language and the ecosystem around it. For example, suppose the developer wants to make a smart contract for the Ethereum ecosystem. The developer has to use language that can be compiled to bytecode which runs on the Ethereum Virtual Machine (EVM), like Solidity. Move language is different as it was designed to be platform-agnostic. That means any blockchain supporting the Move language can run the smart contracts written in it. Projects like Sui and Aptos support the Move language as their smart contract language. [23], [24]

```
1  module address::step_2 {
2      spec module {
3          pragma aborts_if_is_strict;
4      }
5
6      fun sum(first: u64, second: u64): u64 {
7          first + second
8      }
9
10     spec sum {
11         aborts_if first + second > MAX_U64;
12         ensures result == first + second;
13     }
14 }
```

Code listing 4 Move prover specification from [25]

2.3.2 Security

The authors see the security of smart contracts as an existential thread for the widespread use of blockchain technology. Move comes up with *verification friendly design*, and features that would be hard to verify are not included in the language. These include dynamic dispatch, which makes the re-entrance attack impossible, higher-order functions, and exceptions. [26]

Move also comes with having safe math by default, making overflows and underflows impossible in the runtime.

The language also has a bytecode verifier that can check the bytecode before execution to enforce things like no dangling references, acyclic module dependencies, and well-typed procedures and struct declarations. [26]

Not only is the bytecode verified, but there is also a static code analyzer called Move Prover. The developer can use the Move Specification Language (MSL) to specify the properties of smart contracts. Here is an example from [25] documentation examples. The `spec` to create a context where specification can be defined.

The code listing 4 has specifications for a module to make the prover strict, as there is an option to define the exact case when the function aborts. The implementation is just a simple function that adds two numbers. The specifications support pre-conditions and post-conditions when calling the function. The condition defined on line 11 clearly states that the only case when the function aborts. It is when the sum of the two numbers is greater than the maximum value of the unsigned 64-bit integer (overflow).

Programming with Move Prover has similar advantages to Test Driven Development (TDD) as the developer specifies all possible scenarios the function can end with beforehand. When the developer forgets about cases, the prover notifies him that the specification is incomplete.

```
1 module 0x0::BasicCoin {
2     struct Coin has key {
3         value: u64,
4     }
5
6     public fun mint(account: signer, value: u64) {
7         move_to(&account, Coin { value })
8     }
9 }
```

Code listing 5 Move program from the [27]

2.3.3 Abilities

One of the core features of the Move language is to specify abilities on data structures defined using `struct` keyword. The ability is specified using the `struct {name} has {ability}` In code listing 5, the struct has the *key* attribute. Ability *key* allows the type to be used as a key for storage operations. The *store* ability allows other modules or programs to use this type inside its struct definitions. Other developers cannot use the type inside their structs if it is not specifically allowed. Having these abilities is possible only because of the bytecode verifier. Then there is *drop* attribute, which allows the struct to be dropped, and *copy*, that allows the struct to be copied. [27]

The *0x0* on line 1 of Code listing 5 specifies the smart contract's address. The address can later be used for calling the smart contract using its ABI.

When the mint function is run, it saves the Coin struct in the storage. The important part about calling the `move_to` function is that it derives an address in the storage using the signer type. The move runtime automatically passes the transaction sender to the account variable. That means that only this specific keypair can access the data in the storage. Also, there can be only one resource of the same type in the storage. [28]

Having a global state for smart contracts introduces problems of which function accesses specific resources. The Move language solves this problem by using the `acquires` keyword. The `acquires` keyword is used to specify the resources the function will need to access. Having `acquires` is beneficial because the Data Lineage is clearly defined in the code. Also, blockchain implementation can use this information to parallelize the execution of smart contracts. It can find which functions will access different resources and execute them in parallel.

The example of using `acquires` is in Code listing 6, where the function borrowed (acquired a read-only reference) the resource of type `Collection`. The resource is saved under the transaction signer's address. Using the `acquires` can be seen on line 10 after the function return type definition.

The language has a similar concept to the borrow checker in Rust. The developer must always specify the mutability of values and whether the ownership is passed, borrowed, or mutably borrowed. Defining ownership prevents common errors like double spending, as using the value after the ownership is given is impossible. That can be seen in Code listing 7. On line 3, the value was moved to the function, which means it can not be called again. On line 4, the value is already moved. The compiler will throw an error on using the value again.

A unique feature of the Move Language is that it has generics in functions that are callable from the ABI. Smart contracts can therefore store proprietary values passed by other smart

```
module Collection {
    use 0x1::Signer;
    use 0x1::Vector;

    struct Item has store, drop {}
    struct Collection has key, store {
        items: vector<Item>
    }

    public fun size(account: &signer): u64 acquires Collection {
        let owner = Signer::address_of(account);
        let collection = borrow_global<Collection>(owner);

        Vector::length(&collection.items)
    }
}
```

Code listing 6 Acquire keyword from the Move book [28]

```
module 0x0::BasicPay {
    public fun pay(balance: MyCoinBalance) {
        payUsingOtherService(move balance);
        payUsingOtherService(move balance); // compiler error
    }
}
```

Code listing 7 Prevention of double spending

```
module Storage {
    // contents of the box can be stored
    struct Box<T: store> has key, store {
        content: T
    }
}

```

Code listing 8 Generic types in Move from the Move book [28]

contracts. In the Code listing 8, Box struct has a generic type T that is constrained to be allowed to be stored inside a struct.

Chapter 3

Methodology

When comparing blockchain technologies, there are many aspects to consider. As stated in the blockchain trilemma [4], we can only get two out of three of the following: decentralization, security, and scalability.

Performance, one of the critical aspects of blockchain adoption, should not be at the expense of decentralization. After all, that is the differentiating factor compared to traditional bank transactions.

All compared technologies support writing smart contracts in either Rust or Move. When using those languages, the language already provides some security toolings as it has strict rules that the code has to obey. However, blockchain technologies differ in testing frameworks, static analysis tools, and other features that can help developers write secure code.

3.1 Overview of technology and priorities

When new blockchain technology is announced, the maintainers put up a paper with the goals of the technology. This introduction document is called "whitepaper" and often states the differences and problems the technology is trying to solve.

Critical factors from the whitepapers will be summarized in the first section when learning about new technology.

The following criteria will be discussed in the conclusion of the analysis for this section:

- **Primary goals** - The key priorities and goals outlined in the whitepaper in a few sentences.
- **Average daily transactions** - The average daily amount of transactions in the last six months.
- **Average daily active addresses** - The average daily active addresses in the last six months.
- **Mainnet state** - The state of the mainnet

The average data is measured from 2022-10-29 to 2023-04-26 from the artemis.xyz service [29]. The data collection script is available in appendix A.

3.2 Development convenience

Another aspect is the developer experience and the security of the smart contract ecosystem. As the complexity of the code increases, so does the likelihood of defects or errors [30].

The ease of use when writing smart contracts is crucial as it also has its factor in the adoption of the technology. ”Less complex innovations are easier to understand, easier to implement, and less frustrating to use; thus, all else being equal, the lower the perceived complexity, the more likely the innovation will be adopted.” [31].

The following criteria will be discussed in the conclusion of the analysis for this section:

- **Sandbox** - The existence of easy to use web sandbox for writing smart contracts
- **Examples** - Documentation with multiple examples of different smart contracts
- **Programming language** - The programming language that can be used to write smart contracts

3.3 Testing framework

Knowing the consequences of vulnerabilities in smart contracts from [8], the matureness of security tooling for the ecosystem is also important to consider.

Both Rust and Move have a testing framework built into the language. Testing in Move is very similar to Rust as the language took inspiration from some of Rust’s features.

When developing using the TDD it was found there are significant increases in the quality of the code and improvements in overall confidence in the logic [32]. However, considering the defects, research article [33] states that three independent studies conducted that there is no or minor difference between using the TDD and not using it.

Unit tests are designed to test only individual units of code. When defining them, however, a developer can still only define the happy path of his code and not the error cases. It can also provide a false sense of security as the developer can think that the code is written well when it is not.

An example of how a unit test looks like in Move is in Code listing 9. The happy path is in the first test, and the error cases are in the second and third tests. The test on line 13 checks if the language is using SafeMath because the addition of 255 and 1 in an 8-bit unsigned integer type should overflow, and the test should fail.

In Rust, the code is very similar to the Move code. Code listing 10 shows how to define Rust tests. The tests are separated by defining another module and using the `#[cfg(test)]` macro. The math in Rust is also not safe by default, therefore the `checked_add` function is used to check for overflow, and the function returns an `Option<u8>` for a value that could be null addition overflows.

Testing only at the unit level is insufficient as the code can still have defects. The next level of testing is integration testing. The code for integration testing has to be specific to the used blockchain technology. It needs to spin up a testing environment, deploy the smart contract, and run the tests.

After having integration tests in place, it can be beneficial if integration tests can be run with random data. Random data testing is often another feature of a testing framework. It tries to find defects in the code by having a local state and then running public methods of the ABI using random data and trying to find inconsistencies between the local state and the state of the blockchain. Testing with random values injected into the contracts inputs is called fuzz testing.

Having a state space of possible states that can the smart contract be in, fuzz tests can be effective in finding states that are not easily achievable or seen when developing the code or tests.

The existence of unit and integration testing frameworks will be compared in each mentioned technology in the next chapter. The emphasis will be on the integration tests and documentation for using them.

The following criteria will be discussed in the conclusion of the analysis for this section:

- **Unit testing support** - The existence of a unit testing framework

```
module 0x42::Test {
    public fun add(x: u8, y: u8): u8 {
        x + y
    }

    #[test]
    fun addition_test() {
        assert!(add(1, 2) == 3, 1);
    }

    #[test]
    #[expected_failure]
    fun addition_overflow_test() {
        add(255, 1);
    }

    #[test]
    #[expected_failure]
    fun addition_error_test() {
        assert!(add(1, 2) == 4, 1);
    }
}
```

Code listing 9 Simple test in Move

- **Integration testing support** - The existence of integration testing support
- **Fuzz testing** - The existence of fuzz testing framework

3.4 Static analysis

"Static-analysis tools evaluate software in the abstract, without running the software or considering a specific input. Rather than trying to prove that the code fulfills its specification, such tools look for violations of reasonable or recommended programming practices. Thus, they look for places where code might dereference a null pointer or overflow an array." [34]

A static analysis tool tailored to the language and technology is a great feature for developers as it can give hints about the defects. For example, Slither, the code-analysis tool for Ethereum, can detect reentrancy vulnerability exploited in the DAO hack. [35]

When using Rust or Move, the compiler already provides static analysis to a certain level. The compiler, for example, will check for type errors, unused variables, and unused imports. There is no way of dereferencing a null pointer in Rust, as the compiler will not allow it if not using unsafe code.

Static code analyzer is particular to technology and language. The analyzer works best when the developer gives hints about the code. Move has a static analyzer already included with the Move Prover. However, there can be specific features of blockchain technology that the developer must define to Move Prover.

Static analysis tools are not perfect and can have false positives. With too many false positives, the developer may begin to ignore the tool's warnings. Therefore, a tool tailored to language and technology is essential.

The following criteria will be discussed in the conclusion of the analysis for this section:

```rust
pub fn add(a: u8, b: u8) -> Option<u8> {
    a.checked_add(b)
}

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

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

    #[test]
    fn addition_overflow_test() {
        assert!(add(255, 1).is_none());
    }

    #[test]
    fn addition_error_test() {
        assert_ne!(add(1, 2).unwrap(), 4);
    }
}
```

Code listing 10 Example Rust test for addition

- **Static analysis support** - The existence of a static analysis tool and tailored library for a given technology

3.5 IDE support

Good support for the technology in Integrated Development Environment (IDE) can increase productivity at many levels. That is, however, for the cost of learning the new IDE and its features. [36]

When a developer is used to some IDE, the burden of learning a new language can be reduced by having an extension to it. The annual Stack Overflow developer survey from 2022 states that Visual Studio Code (VSCode) remains the preferred IDE among all participating developers. JetBrains-based IDEs for different languages are the second preferred option. [37]

The analysis section will compare the IDE support of the technologies. The existence of extensions for Jetbrains-based IDEs, VSCode or proprietary IDEs will be taken into account.

The following criteria will be discussed in the conclusion of the analysis for this section:

- **VSCode** - The existence of VSCode extension
- **JetBrains IDE support** - The existence of JetBrains IDE extension

3.6 Wallet and software integration

When interacting with the blockchain, the user of the application should not have to worry about the underlying technology. The user experience should be seamless as it would have been a web2 application.

The wallet software is a crucial part of the user experience as it is the first thing the user will interact with. A frontend for interaction with smart contracts is often a web application. All projects provide a JavaScript library for interacting with the nodes. Often it comes with a browser extension that allows the user to create a wallet and manage funds.

The following criteria will be discussed in the conclusion of the analysis for this section:

- **Wallet adapter library** - The existence of a wallet adapter library written by the project
- **Browser extension** - The existence of a wallet browser extension
- **Client library** - The languages the client library for interacting with the blockchain is available

Chapter 4

Analysis of Rust blockchain networks

Rust is a relatively new language having its first public appearance in 2015 [38]. One of the first cryptocurrencies supporting smart contracts (having a Turing Complete language that runs on validation nodes) that is now relatively popular is Ethereum. Being released around the same time as Rust in 2015, [39] it had eight years to mature.

Rust based blockchains with smart contracts were still a new concept, as the programming language they are built upon was released around the same time as Ethereum.

While going through these newly released blockchains, it was not easy to distinguish which ones have the potential to be used in the future. The projects had to show that the proof of concept could be built using it. The choice of contenders was primarily based on the completeness of the documentation. The maturity of the project and funding was also taken into account. The chosen projects are Sui, Aptos, Near, and Solana.

All of the mentioned technologies in the chapter focus on performance, which is the scalability of the blockchain trilemma. The technologies differ on two aspects of the trilemma - decentralization and security. Tradeoffs made on these aspects are discussed in the chapter.

4.1 Sui

Sui is blockchain technology that focuses on low latency smart contracts. Its primary language is its custom flavor of Move programming language that supports the proposed architecture for handling transactions. Sui uses a consensus mechanism called Delegated Proof of Stake (DPoS). The technology uses a native token called Sui with a fixed supply. [40]

Sui is not released yet, only offering devnet and testnet for testing code. The planned launch of its mainnet is in Q1 2023. [41]

When wanting to provide fast transaction times, the blockchain often requires many resources to run a node. Sui solves this problem by using DPoS as an ordinary user does not need to run a full node to participate in the consensus. Compared to ordinary PoS, users vote for a validator node in terms of DPoS called the delegate. When this node is chosen to validate a block, the user gets a reward related to the staked amount in the pool. [42]

The DPoS is often proclaimed to be faster. That is because the number of delegates is capped on some number defined by the protocol. That could lead to centralization as the available delegates can create pools and possibly control the network using the 51% attack.

One of the differentiating features of Sui is low latency, made possible by synchronizing the state of only every epoch. The nodes, instead, all validate and keep track of the transactions locally. It uses the Byzantine Consistent Broadcast protocol to synchronize the network's order and state of transactions. That means the network is only eventually consistent based on the node client requests. [40]

The custom flavor of the Move language is required to parallelize the execution of the smart contracts. It defines two types of storage objects to prevent deadlocks with two transactions accessing the same storage. The first one is owned object that the owner can only modify. The second one is a shared object with no specific owners that more than one user can modify. When the transaction is only accessing the owned objects, it can be executed in parallel with other transactions. [43]

4.1.1 Developer convenience

```
module nfts::shared_auction {
    ...use_statements...
    use nfts::auction_lib::{Self, Auction};

    const EWrongOwner: u64 = 1;

    public entry fun create_auction<T: key + store >(
        to_sell: T, ctx: &mut TxContext
    ) {
        let auction = auction_lib::create_auction(to_sell, ctx);
        auction_lib::share_object(auction);
    }

    public entry fun bid<T: key + store>(
        coin: Coin<SUI>, auction: &mut Auction<T>, ctx: &mut TxContext
    ) {
        auction_lib::update_auction(
            auction,
            tx_context::sender(ctx),
            coin::into_balance(coin),
            ctx
        );
    }

    public entry fun end_auction<T: key + store>(
        auction: &mut Auction<T>, ctx: &mut TxContext
    ) {
        let owner = auction_lib::auction_owner(auction);
        assert!(tx_context::sender(ctx) == owner, EWrongOwner);
        auction_lib::end_shared_auction(auction, ctx);
    }
}
```

Code listing 11 Auction smart contract using shared objects [44]

When saving to Sui storage, the program must define data ownership. The owner can be either an address, where the data can only be used if the owner of this object signs the transaction. Alternatively, it can be an object where the struct is used inside another struct. In the case of the Move language, that can be a generic type specified by the function caller. It can also be an immutable object that cannot be modified after the transaction is executed or a shared object that allows anyone to read or write.

An example of how to interact with Sui storage is Code Listing 11. It is shown on a simple smart contract where an owner of an item wants to sell it. There are two parties - the owner can create an auction, and other users can bid on the item. Then there is a bidder who can bid on items the owner offers in the auction. If the owner is satisfied with offered amounts, they can end the auction, which results in the item being transferred to the highest bidder and cryptocurrency from smart contract to the owner.

The creation of the auction is done using the `create_auction` function seen at line 10 of 11. The `share_object` function is internally calling the `transfer::share_object` function that makes the object accessible to everyone. Bidders can interact with the `bid` function that updates the state of the auction. The `end_auction` is callable only by the owner. That rule is checked on line 29. [45]

The Sui blockchain could not parallelize this implementation as the `Auction` object is shared. The code for auction using owned objects is in Code Listing 12.

When implementing the `Auction` object as an owned object, the bidding can not be done by directly modifying the `Auction` object. There needs to be a separate party that both owner and bidders trust, called the auctioneer, that runs the auction. When creating an auction, the owner specifies the auctioneer. That is called on line 18 of Code Listing 12.

For bidding, another struct `Bid` gets transferred to the auctioneer. The auctioneer then needs to run the `update_auction` function that updates the state of the auction to have the highest bidder.

Auctioneer also ends the auction instead of the owner. There is no need to check if the auctioneer calls the caller the instruction as he owns the data. His private key does not sign the transaction, and the transaction would error at runtime.

As seen from the example auction smart contract, writing smart contracts that can be run in parallel on the nodes is not trivial and can add complexity to the code or introduce more parties that need to be trusted.

4.1.2 Testing framework

The Move testing framework can be used to test the smart contract code. There are helper functions to use the Sui code in the tests. [46]

Even though running a local node for testing is possible, the documentation has no best practices. There is only a mocked version of the interaction with the blockchain instead of checking against an actual running test node.

4.1.3 Static analysis

Sui has a static analyzer in the form of the Move Prover. However, there is no project for something tailored to smart contracts written for the Sui ecosystem.

4.1.4 IDE Support

For developers, they maintain a Visual Studio Code extension that provides syntax highlighting and code completion for the Move language that also supports Sui [47]. For JetBrains IDEs, there is an open-source plugin that supports Move as well [48].

```
module nfts::auction {
    // ...use::statements...
    const EWrongAuction: u64 = 1;
    struct Bid has key {
        id: UID,
        bidder: address,
        auction_id: ID,
        bid: Balance<SUI>
    }

    public fun create_auction<T: key + store>(
        to_sell: T, auctioneer: address, ctx: &mut TxContext
    ): ID {
        let auction = auction_lib::create_auction(to_sell, ctx);
        let id = object::id(&auction);
        auction_lib::transfer(auction, auctioneer);
        id
    }

    public fun bid(
        coin: Coin<SUI>,
        auction_id: ID,
        auctioneer: address,
        ctx: &mut TxContext
    ) {
        let bid = Bid {
            id: object::new(ctx),
            bidder: tx_context::sender(ctx),
            auction_id,
            bid: coin::into_balance(coin),
        };
        transfer::transfer(bid, auctioneer);
    }

    public entry fun update_auction<T: key + store>(
        auction: &mut Auction<T>, bid: Bid, ctx: &mut TxContext
    ) {
        let Bid { id, bidder, auction_id, bid: balance } = bid;
        assert!(object::borrow_id(auction) == &auction_id, EWrongAuction);
        auction_lib::update_auction(auction, bidder, balance, ctx);
        object::delete(id);
    }

    public entry fun end_auction<T: key + store>(
        auction: Auction<T>, ctx: &mut TxContext
    ) {
        auction_lib::end_and_destroy_auction(auction, ctx);
    }
}
```

Code listing 12 Auction smart contract using owned objects [49]

4.1.5 Wallet and software integration

When interacting with the blockchain nodes, there is an SDK for Rust and TypeScript calling the JSON-RPC API of nodes. The TypeScript SDK is still in development. Maintainers of the repository state that there can be breaking changes. [50]

There is also a wallet in the form of a browser extension for Chromium-based browsers and a wallet adapter library for typescript that can interact with the browser extension. However, the library is flagged as experimental. [51]

4.1.6 Summary

A broad summary of the technology that gets later abbreviated in the following Conclusion section 4.5. The following information is provided as of 2023-04-01.

- **Mainnet state** - still in development, release in Q1 2023. [41]
- **Primary goals** - Sui's primary goals are to provide low-latency smart contracts using a custom flavor of the Move programming language and lower the barrier of entry for running a validator
- **Average daily transactions** - unavailable as the mainnet is not live yet
- **Average daily active addresses** - unavailable as the mainnet is not live yet
- **Sandbox** - there is no sandbox, and the developer has to run development locally
- **Examples** - there is only one example smart contract in the repository with an example of tic tac toe [52]
- **Programming language** - custom flavor of Move
- **Unit testing framework** - Move testing framework with provided mocks
- **Fuzz framework** - no fuzz testing framework
- **Integration testing framework** - no integration testing framework shown in the documentation
- **Static analysis support** - Move Prover, no tailored static analysis for smart contracts
- **VSCode** - syntax highlighting and code completion
- **JetBrains IDE support** - syntax highlighting and code completion
- **Wallet adapter library** - experimental library that support two browser-based wallets
- **Browser extension** - provided by the maintainers of the technology, it is called the Sui wallet and is for Chromium-based browsers
- **Client library** - experimental Typescript SDK

4.2 Aptos

Their whitepaper states that Aptos is a blockchain technology designed for scalability, reliability, and safety. It uses the Move programming language as its smart contract language. It also uses a unique approach to transaction processing to achieve parallelization of the execution. It uses its consensus protocol called AptosBFT. Aptos also wants to focus on the extensibility and upgradeability of its protocol. [53]

Aptos offers devnet and testnet for developers to use. The main net was in October 2022 [54].

Compared to Sui, it uses the general Move language and states in the white paper that they do not want to require an upfront knowledge of the data read and written by the developers of smart contracts.

Instead, they came up with scaling the validator performance differently. Transaction processing on the Aptos blockchain is divided into separate stages. Instead of processing transactions right away, Aptos batches transactions into smaller chunks called batches. Then they can be executed concurrently as the runtime can build a transaction dependency graph. [55]

4.2.1 Developer convenience

```
module 0x0::counter {
    use std::signer;

    struct Counter has key {
        count: u64
    }

    public fun get_count(addr: address): u64 acquires Counter {
        assert!(exists<Counter>(addr), 0);
        *&borrow_global<Counter>(addr).count
    }

    public entry fun increment(account: &signer) acquires Counter
    {
        let addr = signer::address_of(account);
        if (!exists<Counter>(addr)) {
            move_to(account, Counter {
                count: 0,
            });
        };

        let counter = borrow_global_mut<Counter>(addr);
        counter.count = counter.count + 1;

    }
}
```

Code listing 13 Counter implemented in Move

Aptos smart contracts are just pure Move code. Therefore there is nothing technology specific to Aptos that developers need to learn. A simple counter in Move language is written in Code

Listing 13. The passed account is initialized using the `move_to` function on line 17. Otherwise, it just takes an account and increments its value.

4.2.2 Testing framework

Aptos supports Unit testing using the Aptos CLI and the testing framework in Move language [56]. Using the Aptos CLI developer can test the smart contracts' internal functions, as the test does not interact with the blockchain. They provide mocked signers and other functions, as seen from the Code Listing 14.

On line 7, the `#[test_only]` attribute is used to mark the module as a test module. The `#[test]` attribute is used to mark the function as a test function. The `counter_account` parameter is used to pass mocked signer to the test function. There is a possibility of getting the signer's address using the `address_of` function. Otherwise, calling the smart contract is straightforward and initializes the account in the first step.

```
 1  #[test_only]
 2  module coin_address::user_coin_tests {
 3      use aptos_framework::account;
 4      use std::signer;
 5      use 0x0::counter;
 6
 7      #[test(counter_account = @0x123)]
 8      public entry fun test_increment(counter_account: signer) {
 9          let addr = signer::address_of(&counter_account);
10          account::create_account_for_test(addr);
11
12          counter::increment(&counter_account);
13          assert!(counter::get_count(addr) == 1, 0);
14
15          counter::increment(&counter_account);
16          assert!(counter::get_count(addr) == 2, 0);
17      }
18  }
```

Code listing 14 Unit test for the Counter smart contract 13

They also support integration testing, which they describe as Transactional testing. The API to use this integration test framework is unique, as they made a specific comment string indicator that can run baseline commands before running functions that make the transactions.

```
1  //# publish [--gas-budget 100]
2  module Alice::first_module {
3      public entry fun foo() {
4          return
5      }
6  }
```

Code listing 15 Transactional testing in Aptos [57]

On line 1 of Code Listing 15, the specific comment string executes a command before running the function. In this case, it is used to publish the module. The command is run with a gas budget of 100.

As of March 2023, the transactional testing feature is still exploratory. Therefore the way of interacting and running the commands before the function can change, and there are only a few examples of its usage. [57]

There is no Fuzz testing library for Aptos as of now. However, in future releases of the transactional testing feature, they could add it as another command for the specific comment string.

4.2.3 Static analysis

Aptos uses the original Move programming language, meaning the project can use any static analyzer being developed in the ecosystem, like the Move Prover.

4.2.4 IDE Support

Move language Aptos uses is supported in VSCode using an extension [47]. For JetBrains IDEs, there is an open-source plugin that supports Move as well [48].

4.2.5 Wallet and software integration

Compared to other blockchain projects, they use REST API instead of JSON-RPC for interacting with the blockchain. They provide SDKs for TypeScript, Python, Rust, and Unity. They also provide a wallet adapter library for TypeScript [58].

There are multiple chromium-based browser wallet extensions, such as Petra Wallet. The documentation [59] claims it is the most commonly used wallet for Aptos-based applications. It exposes the client API through the JavaScript window object for developers to use.

4.2.6 Summary

A broad summary of the technology that gets later abbreviated in the following Conclusion section 4.5. The following information is provided as of 2023-04-01.

- **Mainnet state** - the mainnet was released in October 2022
- **Primary goals** - Aptos aims to provide scalable, reliable, and safe blockchain technology that utilizes the Move programming language. They use a unique transaction processing approach to achieve parallelization while prioritizing extensibility and upgradeability
- **Average daily transactions** - 119506
- **Average daily active addresses** - 19280
- **Sandbox** - there is a community made sandbox [60], that is not mentioned in the documentation.
- **Examples** - there are six examples of simple projects in the documentation for both frontend and backend development
- **Programming language** - Move
- **Unit testing framework** - Move testing framework with provided mocks
- **Fuzz framework** - no fuzz testing framework

- **Integration testing framework** - experimental
- **Static analysis support** - there is Move Prover, but no tailored static analysis for smart contracts to the technology
- **VSCode** - syntax highlighting and code completion
- **JetBrains IDE support** - syntax highlighting and code completion
- **Wallet adapter library** - yes, supporting twelve wallets
- **Browser extension** - multiple chromium-based browser wallet extensions
- **Client library** - Typescript, Python, Rust and Unity SDK

4.3 Near

Near is blockchain technology focusing on the usability and scalability of the smart contracts written in Near. It supports writing smart contracts in every language compiled to Web Assembly (WASM). For the consensus mechanism, Near uses Proof of Stake. [61]

Their mainnet was launched in April 2022 [62]. However, it used Proof of Authority (PoA) as a consensus mechanism. This consensus mechanism does not require the validators to stake tokens, and the malicious validators would not be economically incentivized not to approve invalid transactions. Having managed validator pool is useful when listing authorities that can approve transactions, as it can be cheap to operate. In September 2022, Near switched to Proof of Stake as a consensus mechanism [62].

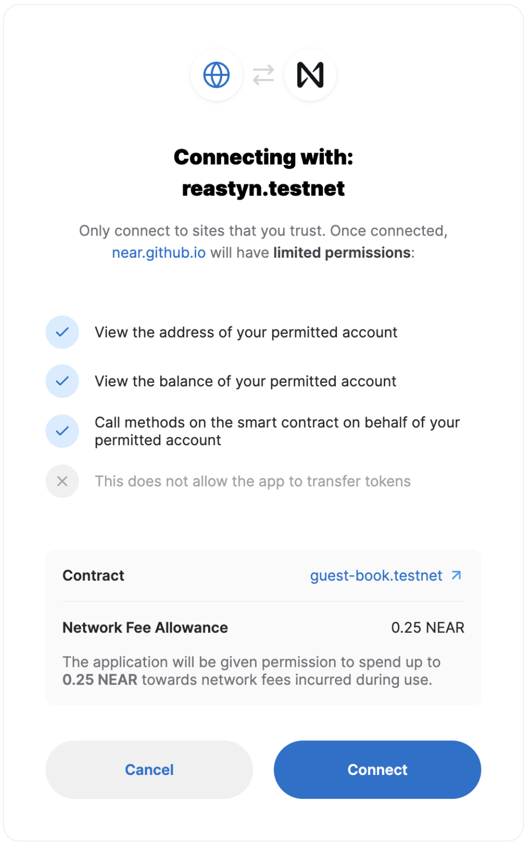
The focus on usability mentioned in the Near whitepaper is segmented into two sections - developer experience and user experience.

For users, the interaction with the blockchain should be as simple as using a web2 application. Their wallet adapter does not require any browser extension to be installed. Instead, it resembles the Single Sign On (SSO) feature of modern web applications. The user is not required to log in using the username and password used specifically for the application. Users can instead use their existing accounts on other services like Google, Facebook, or Twitter. An example of SSO is shown in Figure 4.1.

The documentation did not mention how the protocol works. The protocol mentioned in this section was reverse-engineered from one of the demos they provided in the documentation. When the user opens the application and clicks the login button, the application redirects the user to the web wallet provider (step 2 from Figure 4.2). The application can define which permissions it wants to request from the user and the maximum amount of tokens it can spend. After successful login, the application returns a generated key. The key can be used to interact with the blockchain on behalf of the user. The application can then send a transaction without the user accepting it first.

The protocol for SSO has a tradeoff between usability and security. This generated key is stored in the browser's local storage, which means a malicious actor can steal and use it. For example, when the website would have Cross-site scripting (XSS) vulnerability. Also, the website running the wallet software needs to be up and running for the user to use the application. Even though the user can choose which wallet provider to use, this is centralization in some way, as without the server running, he can not use the application.

The second part of the focus on usability is the developer experience. Because they use WASM on nodes for running the smart contracts, they can offer a wide range of languages to develop. Examples in their documentation cover JavaScript, Rust, and AssemblyScript [63]. They also incentivize developers as when calling a smart contract, 30% of paid gas is given to the smart contract.



Connecting with:
reastyn.testnet

Only connect to sites that you trust. Once connected, near.github.io will have **limited permissions**:

- ✓ View the address of your permitted account
- ✓ View the balance of your permitted account
- ✓ Call methods on the smart contract on behalf of your permitted account
- ✕ This does not allow the app to transfer tokens

Contract	guest-book.testnet ↗
Network Fee Allowance	0.25 NEAR

The application will be given permission to spend up to **0.25 NEAR** towards network fees incurred during use.

Figure 4.1 SSO permission screen

Their second focus is on scalability. They try to achieve good transaction performance by sharding the computational and storage parts. The network can scale by splitting the demand into multiple shards, each running by a different validator pool.

4.3.1 Development convenience

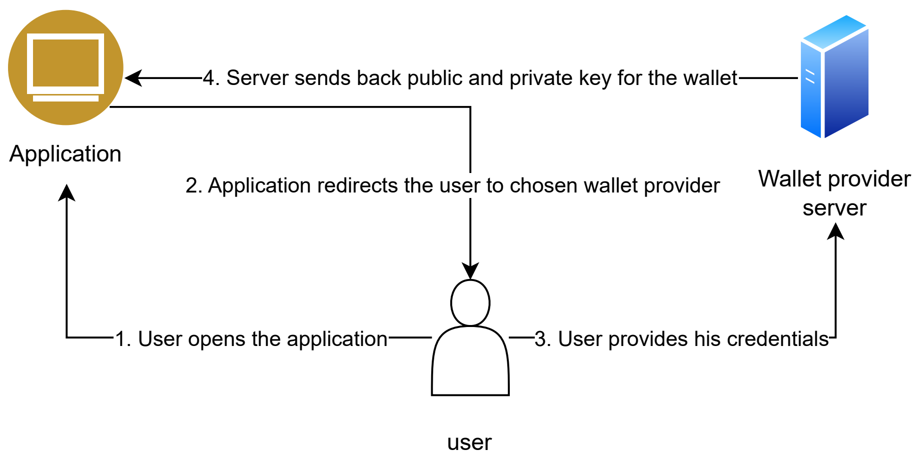
Their documentation has multiple examples of smart contracts having a browser frontend. The frontend is written in javascript. The smart contract language can vary as it could be anything that compiles to WASM.

4.3.2 Testing framework

They use the default Rust testing framework for unit testing and provide common mocking objects for contract initialization. The contract is represented as a struct, and having the internal state defined inside allows making unit tests like there is no blockchain. The example from the introduction chapter Code Listing 2 can be tested as shown in Code Listing 16.

Integration tests are done using the Near Workspaces library for Rust and Typescript. The libraries use Near Sandbox, a running node that can be used for testing purposes. Using the near-workspaces, the developer can set up a running node with a predefined state and deployed contract and run tests against it.

An example of the integration test is shown in Code Listing 17. Before each test, the contract



Application
4. Server sends back public and private key for the wallet
2. Application redirects the user to chosen wallet provider
Wallet provider server
1. User opens the application
3. User provides his credentials
user

Figure 4.2 SSO flow for Near

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn increment() {
        // instantiate a contract variable with the counter at zero
        let mut contract = Counter { val: 0 };
        contract.increment();
        assert_eq!(1, contract.get_num());
    }
}
```

Code listing 16 Unit testing in Near

is freshly deployed, and the state of the blockchain is reset. After each test, the node is stopped. The sandbox also supports time travel which allows testing time-dependent contracts.

The testing framework of Near does not support fuzz tests, and as of the 4th of April, there is no community-made fuzz testing framework for Near.

4.3.3 Static analysis

The BlockSec auditing company developed a static analyzer for NEAR smart contracts called Rustle. It can differentiate 30 problems in smart contracts ranging from info to high severity. The detectors are written in Python or C++. [64]

4.3.4 IDE Support

They are using Rust. Therefore there are IDE plugins that support Rust for both VSCode and JetBrains based IDEs.

```
test.beforeEach(async (t) => {
    const worker = await Worker.init();

    const root = worker.rootAccount;
    const contract = await root.createSubAccount('test-acc');

    await contract.deploy(process.argv[2]);

    t.context.worker = worker;
    t.context.accounts = { root, contract };
});

test.afterEach(async (t) => {
    await t.context.worker.tearDown().catch((error) => {
        console.log('Failed to stop the Sandbox:', error);
    });
});

test('increments the contract', async (t) => {
    const { root, contract } = t.context.accounts;
    await root.call(contract, 'increment');
    const current: string = await contract.view('get_num', {});
    t.is(current, 1);
});
```

Code listing 17 Integration testing of Counter contract

4.3.5 Wallet and software integration

They offer SDKs for Rust and JavaScript for interaction with the blockchain. They also offer a wallet adapter for browser-based and web wallet providers.

4.3.6 Summary

A broad summary of the technology that gets later abbreviated in the following Conclusion section 4.5. The following information is provided as of 2023-04-02.

- **Mainnet state** - the mainnet was released in October 2022
- **Primary goals** - developer and user usability of DApps and scalability
- **Average daily transactions** - 406917
- **Average daily active addresses** - 61277
- **Sandbox** - Gitpod-based sandbox for running and testing smart contracts
- **Examples** - there are twelve examples of projects in the documentation for both frontend and backend development
- **Programming language** - JavaScript, Rust, AssemblyScript
- **Unit testing framework** - default unit testing frameworks with provided Rust and AssemblyScript mocks

- **Fuzz framework -** no fuzz testing framework
- **Integration testing framework -** Near Workspaces for tests written in TypeScript
- **Static analysis support -** Rustle, thirty detectors
- **VSCode -** syntax highlighting and code completion.
- **JetBrains IDE support -** syntax highlighting and code completion
- **Wallet adapter library -** yes, supporting nineteen wallets
- **Browser extension -** multiple chromium-based browser wallet extensions and one provided by maintainers
- **Client library -** JavaScript and Rust SDK

4.4 Solana

Solana is blockchain technology focused purely on high performance. The whitepaper only mentions the proposed technology solutions for low fees and high throughput. It proposed some unique features that can minimize communication between nodes on the network and parallelize the execution of smart contracts. Smart contracts for Solana are written in language that could be compiled to Berkley Packet Filter (BPF) bytecode. It uses the Proof of Stake (PoS) consensus mechanism. [65]

Solana officially launched the mainnet beta in 2020. Only in 2020, after release, there were 8.3 billion transactions. [66]

Communication between nodes and synchronizing state is an expensive operation. Solana solves the problem by using the Proof of History (PoH) as it synchronizes the clock of every node with the option to compute all incoming transactions instead of waiting for the chosen validator to send them. Communication can therefore be replaced by local computation. [65]

Having a local computation, however, can mean that the nodes are not having the same state but have only eventual consistency. Therefore users can not ask every node for the state of the blockchain and achieve the same result.

Replacing communication with local computation still requires having a single validator deciding the next included transactions. That is done using the PoS consensus to the future. When the clocks are synchronized, all nodes know the time of the next slot. That means the validator can be chosen for a specific slot in the future. [65]

The runtime executes the smart contracts in parallel without the need for batching like, for example, Aptos needs to. Instead, all smart contracts are forbidden from having any internal state. A smart contract is just a pure function. If a smart contract has to store some state, it must be passed as an input to this function. The runtime can parallelize the transactions by building dependency graphs of smart contracts accessing the same storage. [67]

4.4.1 Development convenience

Specifying every account used can make developing smart contracts more difficult. In Solana, everything is an account, and there are three types of them - account storing data, programs, and system accounts used by the runtime. [65]

An increment function for the counter smart contract implemented in Near resembles this Code Listing 2. The same smart contract for Solana is shown in Code listing 18. On line 19 of Code listing 18, an entry point to the program upon calling it using a transaction. There is no ABI like in other blockchain networks. Instead, just a byte array is sent to the program. On line 26, the byte array is deserialized using Borsh into the CounterInstruction enum. Therefore the client needs to know how the program serialization logic to serialize the data correctly. On line 28 function is called based on the serialized data.

The process increment function takes the account specified by the user and tries to deserialize it into the Counter struct. After that, the counter is incremented, and the new state is serialized and saved back to the account. The account owner signed the transaction when calling the smart contract. Therefore the smart contract can access the account. There can be a different smart contract, such as decrement the counter, and they can work on the same data account.

An example of calling such a smart contract is shown in Code Listing 19. The first step is to create an account that will be passed to the smart contract. That is on lines 2-12. Then the actual transaction is called while having the account owner as a signer. On line 18, the smart contract sends data about which instruction to call as an enum with only one variant serialized to a single zero byte. The developer must consider what would be used in the future, as adding an instruction can break the client code.

When developing smart contracts in Solana, the developer has more control over what will happen with his smart contract and how it will be executed. There is no ABI like in the technologies specified above. That means that frontend depends on the backend implementation and future releases, as the frontend needs to know how to serialize the data correctly.

Having this amount of control introduces a lot of possible security holes that malicious actors could misuse. A smart contract developer needs always check if the sent accounts are the expected ones, check the transaction's signer, check the accounts' owners, and other operations.

In Code Listing 19, the account was created and passed to the smart contract. The owner of the passed account is the one who called the `create_account` instruction on the system program. There are cases when the program needs to be able to create an account. The programs on Solana are immutable. Therefore it would have no storage where to save the secret key from the created account.

That is the use case for Program Derived Address (PDA). Solana uses the ed2519 elliptic curve algorithm to generate its keys. The elliptic curve can also generate a key that does not lie on the curve. There is only a public key. Instead, the program id is used as proof of account ownership. The system program will only allow writing to the PDA account if it matches the program id. [68]

This amount of control introduces more cognitive load on the developer and can lead to mistakes. That is why the community is developing a framework called Anchor that will help develop smart contracts. It uses Rust macro system to define which accounts and what data needs to be specified.

Having all input data and accounts specified using Rust macros, Anchor can generate Interface Definition Language (IDL). IDL is a language that can specify function names, parameters, and data types [69]. Therefore knowing the meaning of IDL the developer can generate a client-side library for any programming language. Anchor automatically generates Typescript types and can call the IDL specified instructions from the Rust code, making it easier to call the smart contract.

Code Listing 20 shows how to define the same counter smart contract using Anchor. On line 3, the address of this program is defined using the `declare_id!` macro. The program id is later used in other macros like Account to check the account's owner against the program address (public key). First, all accounts required by the increment instruction are defined in lines 22-29. The developer needs to list all accounts with specific properties that allow Anchor to automatically validate the sent account from the user. Therefore, the increment instruction on line 10 automatically serializes the account and allows a mutable borrow.

4.4.2 Testing framework

The developer can use the AnchorProvider from the @project-serum/anchor TypeScript library when testing such smart contracts. Anchor smart contracts automatically generate client-side

```rust
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
};

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub struct Counter {
    pub counter: u32,
}

#[derive(BorshSerialize, BorshDeserialize, Debug)]
pub enum CounterInstruction {
    Increment,
}

entrypoint!(process_instruction);

pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let instruction = CounterInstruction::try_from_slice(instruction_data)?;
    match instruction {
        CounterInstruction::Increment =>
            process_increment(program_id, accounts),
    }
}

pub fn process_increment(
    _program_id: &Pubkey,
    accounts: &[AccountInfo]
) -> ProgramResult {
    let account_into_iter = &mut accounts.iter();
    let state_account_info = next_account_info(account_into_iter)?;

    let mut state = Counter::try_from_slice(
        *state_account_info.data.borrow()
    )?;
    state.counter += 1;
    state.serialize(&mut *state_account_info.data.borrow_mut())?;

    Ok(())
}
```

Code listing 18 Counter smart contract

```typescript
export async function increment(): Promise<void> {
  const counter = Keypair.generate();
  const tx = new Transaction().add(
    SystemProgram.createAccount({
      fromPubkey: payer.publicKey,
      newAccountPubkey: counter.publicKey,
      lamports,
      space: COUNTER_ACCOUNT_SIZE,
      programId
    })
  );

  await sendAndConfirmTransaction(connection, tx, [payer, counter]);

  const instruction = new TransactionInstruction({
    keys: [{ pubkey: counter.publicKey, isSigner: false, isWritable: true }],
    programId,
    data: Buffer.alloc(1),
  });
  await sendAndConfirmTransaction(
    connection,
    new Transaction().add(instruction),
    [payer],
  );
}
```

Code listing 19 Calling the increment function in Typescript

IDE and Typescript types, making testing such contracts easier.

Code Listing 21 shows a sample test. It shared many similarities with the previous calling of increment in Code Listing 19. It uses the AnchorProvider environment to use the IDL generated by Anchor. When reading from the account, there is no need for interaction with the smart contract. It reads directly from the account. When using the anchor library, serialization is unnecessary, as seen in the 34th line where the account is queried from the blockchain.

Unit testing is more challenging in Solana than in other blockchain technologies, as the program can not contain any internal state. When giving accounts to the program, they must be mocked and serialized using Borsh. What is done instead is that the core logic is extracted into a separate function and tested using Rust unit tests.

It is possible to use TypeScript and the Solana SDK for integration testing. The skeleton tests are automatically generated if the developer uses the Anchor framework. It uses the Mocha testing framework for Typescript and Javascript.

Even though there is an integration testing framework, as of the 4th of April, no fuzz testing library is available for Solana.

4.4.3 Static analysis

There are multiple static analyzer tools for Solana. VRust has eight detectors for common errors in Solana smart contracts like Missing owner or signer checks and possible integer overflows [70]. The sec3 auditing company also provides Soteria, a paid static analysis tool for Solana smart contracts [71].

```rust
use anchor_lang::prelude::*;

declare_id!("7raN7YsohXGg91o7ZATJ8GKp7bEmrEZtxbkt5MkgiiLn");

#[program]
pub mod anchor_counter {
    use super::*;

    pub fn increment(ctx: Context<Update>) -> Result<()> {
        let counter = &mut ctx.accounts.counter;
        counter.count += 1;
        Ok(())
    }
}

#[account]
pub struct Counter {
    pub count: u64,
}

#[derive(Accounts)]
pub struct Update<'info> {
    #[account(init_if_needed, payer=user, space = 8 + 8)]
    pub counter: Account<'info, Counter>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

Code listing 20 Counter smart contract using Anchor

4.4.4 IDE support

It uses Rust for writing smart contracts. There are plugins for both VSCode and Jetbrains-based IDEs. A plugin for Anchor specifically allows interaction with the Anchor CLI and provides code completion for the Anchor framework. [72]

4.4.5 Wallet and software integration

When interacting with Solana smart contracts, clients do not need any wallet installed. Accounts can be queried without any interaction with smart contracts. That is good, as when the frontend is a website, it does not need any browser extensions to be installed. However, the user needs a wallet when modifying the state of any account. As of the 4th of April, multiple wallets are available for Solana. Phantom is the most popular option for browser extensions based on [73]. There is also a browser called Brave that has a built-in Solana wallet.

A wallet adapter library for developers allows users to choose from 48 different wallets for the browser.

```
describe("anchor-counter", () => {
  const provider = anchor.AnchorProvider.env();
  anchor.setProvider(provider);

  const program = anchor.workspace.AnchorCounter as Program<AnchorCounter>;
  const payer = provider.wallet;
  const counter = anchor.web3.Keypair.generate();

  it("Initialize the account for later use", async () => {
    const counter = Keypair.generate();
    const lamports = await provider.connection
        .getMinimumBalanceForRentExemption(COUNTER_ACCOUNT_SIZE);
    const tx = new Transaction().add(
      SystemProgram.createAccount({
        fromPubkey: provider.wallet.publicKey,
        newAccountPubkey: counter.publicKey,
        lamports,
        space: COUNTER_ACCOUNT_SIZE,
        programId: program.programId,
      })
    );

    await provider.sendAndConfirm(tx, [counter]);
  });

  it("Increment state of that account", async () => {
    const tx = await program.methods
      .increment()
      .accounts({ counter: counter.publicKey, user: payer.publicKey })
      .signers([counter])
      .rpc();

    const account = await program.account.counter.fetch(counter.publicKey);
    expect(account.count.toNumber() === 1);
  });
});
```

Code listing 21 Integration test of the Anchor Counter

4.4.6 Summary

A broad summary of the technology that gets later abbreviated in the following Conclusion section 4.5. The following information is provided as of 2023-04-04.

- **Mainnet state** - the mainnet was released in Beta in February 2020
- **Primary goals** - high performance and low transaction fees
- **Average daily transactions** - 20640569
- **Average daily active addresses** - 364453
- **Sandbox** - yes, custom solution for Solana called SolPG [74]

- **Examples** - there are multiple examples of both frontend and backend code varying in complexity [75]
- **Programming language** - Rust
- **Unit testing framework** - not any mocks, Rust unit testing framework could be used, but code must be written in a specific way
- **Fuzz framework** - no fuzz testing framework
- **Integration testing framework** - yes, Anchor build with TypeScript and Trdelnik build in Rust
- **Static analysis support** - yes, VRust and Soteria
- **VSCode** - syntax highlighting and code completion
- **JetBrains IDE support** - syntax highlighting and code completion.
- **Wallet adapter library** - yes, supporting 48 wallets
- **Browser extension** - multiple browser wallet extensions
- **Client library** - JavaScript and Rust SDK

4.5 Conclusion

In conclusion, all of the technologies mentioned in this chapter have tradeoffs in features they offer to the developer regarding ease of development, tooling for testing the code, and development tooling.

4.5.1 User Usability

All the mentioned projects require the user to use some browser wallet to interact with the smart contract. While they allow the user to read from the blockchain without any wallet, it is still required to use a wallet to modify the state of the blockchain. That includes Ethereum, which also requires the user to use a wallet to interact with the smart contract.

Near is the only project that allows the user to interact with the blockchain without any wallet. The technology of having multiple keys with different permissions is a feature unique to Near. It would be a great use case for applications that need to run on browsers that do not support browser extensions or wallets that are not yet developed.

4.5.2 Developer usability

Solana requires the developer to specify each account used in the smart contract making the smart contract immutable and having no internal state. Consequently, it needs to define additional logic like checking the ownership of the account, serializing and deserializing the data, or checking that the passed account is valid.

Sui and Aptos require the developer to consider which accounts will be acquired by the smart contract and which will be passed to it. Compared to Solana, they do not need to serialize and deserialize the data and check the owner as it is done by the runtime automatically.

Sui adds more cognitive load than Aptos because the developer needs to think about the type of storage for his data (shared objects vs. owned objects).

When knowing the Move programming language, it would be easier to opt for Aptos as it uses the native language. Near, being written in Rust, has to use macros to implement specific features for smart contract development native to Move.

Near has similar internal state management to Ethereum. The user can mutate the smart contract's state without specifying the accounts beforehand. Developing smart contracts without considering where the internal state is stored is easier. That comes at the cost of being unable to parallelize the transactions, as any transaction can modify the state.

4.5.3 Unit tests

Unit testing is possible in all of the mentioned projects. Both Sui and Aptos can use the Move testing framework. Sui provides some mocking capabilities allowing one to get higher code coverage without interacting with the blockchain [46]. Aptos provides a similar mocking framework as well [76].

Near uses the Rust testing framework and also provides unit testing mocking capabilities. [77]

Solana is the only project that does not provide any mocking of the blockchain. It would be more problematic as it would have to mock not only the accounts and mutability but also the verification of the accounts.

Ethereum has multiple testing frameworks supporting unit testing, including Truffle [78] and Brownie [79].

4.5.4 Integration tests

Sui only offers a way of running mocked transactions instead of running them on a local test node. No best practices are mentioned in the documentation for integration testing in Sui.

Aptos provides an integration testing framework that uses comments to specify the state of the blockchain before and after calling the transaction. It is only in an exploratory phase that breaking changes can occur. [57]

Near and Solana provide integration testing frameworks using Typescript SDK and testing frameworks. They both provide a library for spinning up a local node and running the tests against it.

Ethereum provides multiple integration testing tools, including Ganache [80] and Hardhat [81].

4.5.5 Fuzz tests

Even though Sui is the only project that does not have an integration testing framework, no fuzzing framework is available for any of the projects.

Ethereum has an edge by having multiple fuzz frameworks like Echidna [82] and Woke [83].

4.5.6 IDE plugin and dev tooling

The projects provide IDE plugins for VSCode and JetBrains-based IDEs. Sui maintainers contribute directly to the Move language plugin for VSCode.

When using Solana with Anchor, a plugin for VSCode allows interaction with the Anchor CLI and provides code completion. [72]

Ethereum also supports VSCode [84] and JetBrains-based IDEs [85].

4.5.7 Static analyzer

Sui and Aptos both use the Move language having the option to use Move Prover for static analysis. Other than the Move Prover, Move has no vulnerability detection tool.

Solana has multiple options to choose from. There is a paid static analysis tool called Soteria [71], and VRust, which was proposed in the paper but was not released to the public yet [70].

Near has a static analysis tool developed by the BlockSec auditing company called rustle [64].

Ethereum is more mature with Slither [86] having over 84 different detectors with impact levels from optimization to high.

4.5.8 Wallet software and integration

All projects have multiple wallets available for the browser. They also implement some form of wallet adapter library that the project's developers maintain. [58] [87] [88] [89]

Near being the only project inventing the concept of multiple keys with different permissions has a wallet that allows changing the state of the blockchain without any wallet.

4.5.9 Summary

The summary Table 4.1 compares all mentioned projects in this chapter. It outlines only the most essential features. For more information, see the individual sections.

When looking only at statistics of the last six months, Solana has the highest average transactions and active addresses. Near having four times the Aptos transactions could result from the Rainbow Bridge, which allows the Near blockchain to interact with the Ethereum blockchain.

Solana was released two years earlier than both Aptos and Near. That could be why the average transactions and active addresses are higher than competitors. However, the difference in the number of transactions is significant, with Solana having fifty times more transactions than Near and six times more active addresses.

Sui is the only project not providing only based IDE for the developers to try out developing smart contracts.

For programming languages, Near has only listed Javascript, Rust, and AssemblyScript as they show support in the documentation. However, using any language that compiles to WASM is possible.

Solana is the only project that does not show the unit testing in its docs nor provides a mocking library.

Sui and Aptos both use the Move language. Therefore they can use Move Prover for static analysis. Solana has multiple options with a static analysis tool called Soteria and VRust. Near has an open-source static analysis tool developed by the BlockSec auditing company called Rustle.

All projects have good IDE support.

Wallet adapter libraries are available for all projects. Near is the only project that does not require a browser extension wallet to interact with the blockchain.

Client libraries are available for all projects. Aptos being the only project supporting Python and Unity for their SDK.

	Sui	**Aptos**	**Near**	**Solana**
Mainnet state	No	Yes, released in October 2022	Yes, released in October 2022	Yes, in Beta, released in February 2020
Primary goals	Low latency smart contracts with low barrier of entry for validators	Parallelized execution of the smart contract execution while prioritizing extensibility and upgradeability	Developer and user usability of DApps and scalability	High performance and low transaction fees
Average daily transactions	No mainnet	119506	406917	20640569
Average daily active addresses	No mainnet	19280	61277	364453
Sandbox	No	Yes, Pontem	Yes, Gitpod	Yes, SolPG
Project examples	1	6	12	12
Programming language	Sui Move	Move	JavaScript, Rust, AssemblyScript	Rust
Unit testing framework	Move testing framework with provided mocks	Move testing framework with provided mocks	Mocks provided Rust and AssemblyScript testing frameworks	No
Integration testing framework	No	Experimental	Yes	Yes
Fuzz framework	No	No	No	No
Static analysis support	Move Prover, nothing tailored to Sui	Move Prover, nothing tailored to Aptos	Rustle	Soteria and VRust
VSCode	Yes	Yes	Yes	Yes
JetBrains IDE support	Yes	Yes	Yes	Yes
Wallet adapter library	Experimental	Yes, 12 wallets supported	Yes, 19 wallets supported	Yes, 48 wallets supported
Browser extension	Yes, maintained by the project	Yes, multiple community made extensions	Yes, multiple community made extensions and project maintained one	Yes, multiple community made extensions
Client library	Typescript SDK	Typescript, Python, Rust and Unity SDK	JavaScript and Rust SDK	JavaScript and Rust SDK

Table 4.1 Comparison table of blockchain technologies

Chapter 5
Contribution to security tooling

When going through the conclusion of the previous chapter, there were missing tools specifically for fuzz testing. Blockchain technology must implement integration testing to implement a fuzz testing framework on top of it.

Requirements for integration testing framework narrowed down the scope of blockchain technologies as Aptos has integration tests in the exploratory phase, and Sui does not mention them in their documentation.

The possible technologies to choose from are, therefore, Solana and Near. The supervisor of this thesis is a co-founder of a company developing a testing framework for Solana. The testing framework is Trdelnik, and it lacks the fuzzing feature.

The contribution to security tooling will be to improve the Trdelnik testing framework. It will be upgraded to the newest versions, the documentation will be improved, and support for writing fuzz tests will be added. The code for these improvements can be found in Appendix A.

5.1 Fuzz testing

There are multiple types of fuzz testing. Black box fuzz tests randomly generate bytes as input and check if the program can handle them. Black box fuzz testing can be helpful for programs retrieving files as inputs. Just try randomly generating bytes and alert if the program crashes.

Gray box fuzz tests that have some knowledge about the program. The developer must know the interesting or invalid states of the program and how to check them programmatically.

Then there is white box testing, where the tester knows the structure and functions of the program and can generate inputs that are more likely to cause errors. [90]

When fuzz testing a smart contract, the testing result should be getting the contract into a state that the programmer did not intend. An unintended state is not necessarily an error. However, it can be helpful information about the contract as it can potentially lead to a vulnerability as the developer did not intend for anyone to enter this state.

Multiple companies discovered vulnerabilities or bugs using fuzz testing. Namely, using fuzz testing, Microsoft discovered more than 30 bugs in their software [91]. Alternatively, Google reported 1000+ bugs (including 200+ vulnerabilities) in open source projects [92].

For smart contracts, white box fuzz testing is more suitable as the tester knows the structure of the contract and can generate inputs that are more likely to cause errors.

The fuzz testing can be implemented by having some tester-defined state that gets compared with the actual state of the contract. If the states are different, the tester must be alerted. The tester can then decide if the state is unintended or not. If the state is unintended, the tester can investigate it further.

```
state = initial_state()

while not done():
    random_function = rand_choose(contract.functions)
    mutated_state = mutate_state(state, random_function)
    if is_unintended_state(mutated_state):
        alert_user(mutated_state)
        break

```

Code listing 22 Pseudocode of smart contract fuzzing

When applied to smart contracts, the tester creates a state that holds the information about what the contract should have. Then the tester runs different functions on the contract and checks if the state corresponds to the expected state. If the state is different, the tester can investigate it further.

The Code listing 22 shows a pseudocode of how the fuzz testing is implemented.

Before developing the API for the fuzz tests, there are multiple already existing fuzzing libraries to inspire from. For Ethereum, there is the Echidna and Woke fuzzing frameworks.

Echidna is a grey box testing framework as it can parse the ABI of the contract and generate inputs that are more likely to get desired unintended state. The developer specifies invariants, which are functions to check if the contract is in the correct state or has reached an unintended state. [93]

Woke implements the white box fuzz testing. It defines two types of interacting with the smart contract. ”Flows” are functions mutating the state of the smart contract. The ran flow is randomly chosen from the array of registered flows. Then ”invariants” check the current state of the smart contract. All of the invariants are run after flow. [83]

The pseudocode for white box fuzz tests is shown in Code listing 23.

```
state = initial_state()

while not done():
    random_flow = rand_choose(flows)
    mutated_state = mutate_state(state, random_flow)
    for invariant in invariants:
        if invariant(mutated_state):
            alert_user(mutated_state)
            break
```

Code listing 23 Pseudocode of white box fuzz testing

For this thesis, similar white box fuzz testing will be implemented for the Solana testing framework Trdelnik.

5.2 Upgrade of Trdelnik

Upon starting the implementation of the fuzz testing, Trdelnik was not supporting Anchor 0.27.0. The first task of this thesis was to upgrade Trdelnik to support the latest version of Anchor and Solana.

There were multiple changes introduced to the Anchor client. The Anchor Client used for interacting with the blockchain was now accepting any generic type for the config.

All Solana dependencies were updated to the 0.15.2 version. There were some new RPC methods available that were added to Trdelnik Explorer.

5.3 Improvements for the next version of Trdelnik

To test out Anchor, Solana, and the Trdelnik framework, a smart contract implementing the D21 voting method was implemented. Tests were written in the default Anchor testing framework in Typescript and Trdelnik. When running the Anchor tests, the tests were running for around 23 seconds. When running Trdelnik tests, they were running for 2 minutes and 3 seconds. The chapter 6 will detail the time it takes to run the tests.

Running the Trdelnik tests took five times more than running the Anchor tests. Nonetheless, they cover the same amount of code. The time comparison is unfair as the Trdelnik tests are running on a freshly started test Solana node, whereas the Anchor tests are reusing the same Solana node. Anchor tests make the developer write tests a certain way, as resetting a smart contract's state is impossible.

Anchor tests start the Solana test validator node when the testing runtime starts and shut them down after the tests are finished. Trdelnik tests spin up a new Solana test validator for each test.

Having the tests run on a new Solana node is a good thing, as it ensures that the tests are not dependent on the state of the blockchain. However, the tests are taking too long to run.

In order for the tests to run faster, they need to run in parallel. There are two possible ways of having a separation of concerns while still having the tests run reasonably fast. Deploy the smart contract to different addresses on the same node or spin up multiple test validator nodes.

5.3.1 Multiple smart contracts with different addresses

Deploying smart contracts on different addresses is a common way of parallelizing tests. Smart contracts are separated as they only share the running test validator node, not the state of the contract.

The testing framework deploys the smart contract to a new address for each test. The tests are querying the same node but requesting different addresses based on the thread.

The Anchor framework uses the `declare_id!` macro as mentioned in 20. The program id is compiled into the smart contract's bytecode, which makes deploying the same smart contract to different addresses impossible.

5.3.2 Multiple test validator nodes

Another option for running the tests in parallel is to spin up multiple test validator nodes. The framework will spin up a new test validator node for each test and shut it down after it is finished.

Trdelnik and Anchor run the `solana-test-validator` executable. When installing Solana Tool Suite, it is automatically added to `PATH` variable. Using the CLI limits the configuration options to what is implemented in the CLI options.

The test validator is implemented in Rust as part of the Solana project [94]. The source code for starting the validator can be used as an example of starting a custom validator with specified options.

This option is more flexible as it allows the developer to specify the configuration of the test validator. The developer can pre-deploy the smart contract and accounts to the test validator, further speeding up the tests.

To further speed up the tests, the developer can specify the number of threads to run the tests on. The tests can be run on multiple threads, each thread running a test validator node. The tests can be run in parallel, each thread running a test validator node.

This is how the test for the increment counter smart contract from 21 is implemented in Trdelnik after having the option to create a validator per thread.

```rust
1  struct Fixture {
2      client: Client,
3      increment_account: Keypair,
4  }
5  impl Fixture {
6      fn new(client: Client) -> Self {
7          Fixture {
8              client,
9              increment_account: keypair(1),
10         }
11     }
12
13     #[throws]
14     async fn deploy(&mut self) {
15         self.client
16             .airdrop(self.client.payer().pubkey(), 5_000_000_000)
17             .await?;
18     }
19 }
```

Code listing 24 Fixture for increment counter test

First, Trdelnik allows developers to create fixtures for the test that can be reused. The fixture is in Code listing 24. The fixture initializes all accounts that would be required in the test itself. The fixture gets passed to a client connected to the test validator node. The keypair for an account that stores the counter is created on line 9. The client is used to airdropping the account on line 10. The airdrop is required as the account is not funded by default.

```rust
1  #[throws]
2  #[fixture]
3  async fn init_fixture() -> Fixture {
4      let mut validator = Validator::default();
5      validator.add_program("anchor_counter", PROGRAM_ID);
6      let client = validator.start().await;
7
8      let mut fixture = Fixture::new(client);
9      fixture.deploy().await?;
10     fixture
11 }
```

Code listing 25 Fixture initialization for the test

In Code listing 25 the Fixture is initialized. The developer can use the `#[fixture]` macro for asynchronously initializing the fixture. Here the developer uses the `Validator` instance to create the test node validator as seen on line 4. The creation of a validator is done using the builder pattern. After the configuration is specified, the developer can call the `start` method to start the validator. The validator returns a client that can be used to interact with the blockchain.

In Code listing 26 is the calling of the test itself. The test function is marked with the `#[trdelnik_test]` macro which automatically handles the parallel executions of the tests. It uses the `#[future]` macro for injecting the fixture using the `init_fixture` function. After the fixture initialization, the test is written similarly to its Anchor test counterpart.

```rust
#[trdelnik_test]
async fn test_increment(#[future] init_fixture: Result<Fixture>) {
    let fixture = init_fixture.await?;

    anchor_counter_instruction::increment(
        &fixture.client,
        Increment {},
        Update {
            counter: fixture.increment_account.pubkey(),
            user: fixture.client.payer().pubkey(),
            system_program: System::id(),
        },
        vec![
            fixture.client.payer().clone(),
            fixture.increment_account.clone(),
        ],
    )
    .await?;

    let counter_account = fixture
        .client
        .get_account(fixture.increment_account.pubkey())
        .await?
        .unwrap();

    let counter_account = Counter::try_deserialize(
        &mut counter_account.data()
    ).unwrap();
    assert!(counter_account.count == 1);
}
```

Code listing 26 Test of the increment counter smart contract

Implementing the validator had multiple challenges as it uses system ports for communication. The ports are used for the RPC, gossip protocol, and database. The ports are chosen randomly and they wait to check if the ports are still available to prevent the program from panicking from trying to start on an already-used port.

The other problem was storing the state of the blockchain. That is solved by using a temporary directory with a randomly generated folder name as well. The temporary directory is deleted if the test is finished with success. Otherwise, the directory is kept for debugging purposes.

5.3.3 Improvement of the code generator

Trdelnik test framework automatically generates functions for calling the smart contract instructions from the IDL. Anchor generates the IDL and is later used by Trdelnik to generate the test functions.

When Trdelnik generated the functions it would flatten the hierarchy of the parameter passing. Code listing 27 shows calling a more complex instruction. Calling the function relies on IDE to provide the names for the parameters, and the function's calling is not very readable.

```rust
pub async fn vote(
    client: &Client,
    i__voter_bump: u8,
    i__subject_bump: u8,
    i__basic_info_bump: u8,
    i_subject: Pubkey,
    i_is_positive_vote: bool,
    a_voter: anchor_lang::solana_program::pubkey::Pubkey,
    a_subject: anchor_lang::solana_program::pubkey::Pubkey,
    a_basic_info: anchor_lang::solana_program::pubkey::Pubkey,
    a_initializer: anchor_lang::solana_program::pubkey::Pubkey,
    a_system_program: anchor_lang::solana_program::pubkey::Pubkey,
    signers: impl IntoIterator<Item = Keypair> + Send + 'static,
) -> Result<EncodedConfirmedTransactionWithStatusMeta, ClientError>
{...implementation}
```

Code listing 27 Generated function for calling the vote instruction

Instead, Trdelnik now generates the functions using the structs defined in the instruction, making the calling of the function more readable, and the order of passed accounts or parameters does not matter. As seen in Code listing 28, the function is generated using the structs defined in the IDL.

```rust
pub async fn vote(
    client: &Client,
    parameters: d21::instruction::Vote,
    accounts: d21::accounts::Vote,
    signers: impl IntoIterator<Item = Keypair> + Send + 'static,
) -> Result<EncodedConfirmedTransactionWithStatusMeta, ClientError>
{...implementation}
```

Code listing 28 Generated function for calling the vote instruction using structs

Calling such functions look like in Code listing 29. The developer has to specify the parameters and accounts in the correct order.

```
d21_instruction::vote(
    &voter.client,
    voter.account.bump,
    subject.subject.bump,
    common.basic_info.bump,
    subject.client.payer().pubkey(),
    positive_vote,
    voter.account.pubkey(),
    subject.subject.pubkey(),
    common.basic_info.pubkey(),
    voter.client.payer().pubkey(),
    System::id(),
    Some(voter.client.payer().clone()),
)
```

Code listing 29 Calling the vote instruction using the generated function

This is how the call looks like when using the structs 30. The order of the parameters and accounts does not matter. The IDE automatically recommends specified parameters inside of the structs.

```
d21_instruction::vote(
    &voter.client,
    d21::instruction::Vote {
        subject: subject.client.payer().pubkey(),
        is_positive_vote: true,
        voter_bump: voter.account.bump,
        subject_bump: subject.subject.bump,
        basic_info_bump: common.basic_info.bump,
    },
    d21::accounts::Vote {
        voter: voter.account.pubkey(),
        subject: subject.subject.pubkey(),
        basic_info: common.basic_info.pubkey(),
        initializer: voter.client.payer().pubkey(),
        system_program: System::id(),
    },
    Some(voter.client.payer().clone()),
)
```

Code listing 30 Calling the vote instruction using the generated function using structs

5.4 Implementation of fuzz tests

The implementation started by designing an Application Programming Interface (API) that the developer will interact with. The first design was based on the Woke testing framework. This framework is written in Python and looks like the Code listing 31.

```python
class TestingSequence:
    def __init__(self, contract: IncrementContractType):
        self.owner = random_account()
        self.contract = contract.deploy({"from": self.owner})
        self.current_value = 0

    @flow
    def increment(self):
        self.current_value += 1
        self.contract.increment({"from": self.owner})

    @invariant
    def check_value(self):
        assert self.contract.get() == self.current_value

def test_campaign(voting_contract: IncrementContractType):
    campaign = Campaign(lambda: TestingSequence(IncrementContractType))
    campaign.run(1000, 400)
```

Code listing 31 Fuzz test API in Woke

In Woke, the functions prefixed with `test_` are treated as tests to run. It creates an instance of Campaign, which takes a class as its parameter. On the class, the developer can mark functions with `@flow` and `@invariant` decorators. The `@flow` decorator marks a function as a flow of the test. The `@invariant` decorator marks a function as an invariant of the test. The invariants are checked after each flow. The fuzz tests start from scratch 1000 times, and 400 flows run in each test. The testing will stop if the assert in invariant fails. [83]

Multiple problems exist when mimicking the Python3 API in Rust. Rust is a statically typed language. Even when the macros can be used on methods for struct, there is no way of dynamically registering them as functions that would be dynamically called. The other problem is that the functions are called in parallel and will block when accessing the same state.

5.4.1 API requirements

The developer should not need to worry about parallelization and should be able to write sequential code that does not require using mutexes or other synchronization primitives.

The developer should be able to specify struct with the local state to compare with the one on the blockchain.

The developer should be able to specify flows with mutable access to the local state and client to interact with the blockchain.

The developer should be able to specify invariants with readable access to the local state and client to interact with the blockchain.

5.4.2 API design

The builder pattern is a common way to deal with such APIs in Rust. The final design of the API looks like Code listing 32.

```rust
struct TestState {}

async fn main() {
    FuzzTestBuilder::new()
        .initialize_validator(initialize_fn)
        .add_flow(flow_fn)
        .with_state(TestState {})
        .add_init_handler(init_contract)
        .add_invariant(invariant_fn)
        .start(n_seq, n_flows)
        .await;
}
```

Code listing 32 The final design of the fuzz test API

Multiple functions can be called on the builder. Examples of such functions will be shown on an extended smart contract from 20 that also supports the decrement function.

The `initialize_validator` function takes a function that initializes the validator. The handler needs to return the `trdelnik_client::validator::Validator` struct. An example of this function used for the counter smart contract is shown in Code listing 33.

Then there is the `init_contract` that can register callbacks which are run before the fuzzer tests begin with deployed client. It can interact with the blockchain and initialize the contract or airdrop tokens to testing accounts. An example is on lines 7-9 in Code listing 33.

```rust
1  fn initialize_validator() -> Validator {
2      let mut validator = Validator::default();
3      validator.add_program("anchor_counter", PROGRAM_ID);
4      validator
5  }
6
7  async fn init_contract(client: Client) {
8      client.airdrop(client.payer().pubkey(), 5_000_000).await;
9  }
```

Code listing 33 The initialize validator function for the counter smart contract

The `with_state` function takes a struct that represents the local state. An example of this function used for the counter smart contract is shown in Code listing 34.

```rust
#[derive(Clone, Debug)]
struct TestState {
    count: i128,
    counter_account: CopyableKeypair,
}

fn main() {
    let state = TestState {
        counter_account: CopyableKeypair(Keypair::new()),
        count: 0,
    };
    FuzzTestBuilder::new()
        .with_state(state);
}
```

■ **Code listing 34** The with_state function for the counter smart contract

The `add_flow` function takes a handler representing the fuzz test's flow. The `add_invariant` function takes a handler representing an invariant of the test. Both handlers can specify which parameters they need just using the function's parameters. The function can specify the parameters in any order, and they get automatically mapped to the correct parameter. Usually, mapping parameters would not be possible in compiled languages. How this is done is explained in the following subsection.

An example of this function used for the counter smart contract is shown in Code listing 35.

```rust
async fn flow_increment(State(mut state): State<TestState>, client: Client) {
    // Make blockchain operation
    anchor_counter_instruction::increment(
        &client,
        Increment {},
        Update {
            counter: state.counter_account.0.pubkey(),
            user: client.payer().pubkey(),
            system_program: System::id(),
        },
        vec![client.payer().clone(), state.counter_account.0.clone()],
    )
    .await
    .unwrap();
    // Synchronize local state
    state.count += 1;
}
```

■ **Code listing 35** The add_flow for the counter smart contract

Similarly, there is an example of the `add_invariant` function in Code listing 36.

```rust
async fn invariant_check_counter(
    client: Client,
    State(state): State<TestState>,
) {
    let counter_account = client
        .get_account(state.counter_account.0.pubkey())
        .await
        .unwrap()
        .unwrap();

    let counter_account = Counter::try_deserialize(
        &mut counter_account.data()
    ).unwrap();
    assert_eq!(counter_account.count as i128, state.count);
}
```

Code listing 36 The add_invariant for the counter smart contract

The `start` function takes the number of sequences and the number of flows to run. The `start` function returns a future is awaited for the fuzz tests to block until the number of sequences and the number of flows is reached or the assert in invariant fails.

The whole fuzz test would then look like Code listing 37.

```rust
FuzzTestBuilder::new()
    .initialize_validator(initialize_validator)
    .add_flow(flow_increment)
    .add_flow(flow_decrement)
    .with_state(TestState {
        counter_account: CopyableKeypair(Keypair::new()),
        count: 0,
    })
    .add_invariant(invariant_check_counter)
    .start(2, 250)
    .await;
```

Code listing 37 Calling fuzz test for the counter smart contract

5.4.3 Arity functions

With the limitation of only accepting functions, the fuzz tester must accept a function with variadic parameters, as the developer needs to be able to change the local state. It also needs to be possible to check this state inside the invariants.

Rust has a concept of traits for abstractly defining shared behavior. The structs are similar to interfaces in other languages. They are often used to define a set of methods that must be implemented for a struct. [95]

The difference to interfaces in other languages is that in Rust, any type can implement a trait, including functions. [96]

The `add_flow` method in the fuzz builder looks like Code listing 38. The generics are used to define the parameters of the function. The passed function of generic type `F` need to implement the `Handler` trait with a variable amount of arguments.

```rust
pub fn add_flow<F, Args>(&mut self, flow: F) -> &mut Self
where
    F: Handler<Args> + 'static + Sync + Send,
{
    if self.started {
        panic!("You cannot add flows after the `start` method was called.");
    }
    self.add_handler(self.flows.clone(), flow);
    self
}
```

Code listing 38 The add_flow method in the fuzz builder

The `Handler` trait is defined in Code listing 39. When passing the function, Rust automatically adds this call to the function. That function receives the Mutex guard of the saved local state and returns a future.

```rust
pub trait Handler<T>: Clone + Send + Sized + 'static {
    type Future: Future<Output = ()> + Send + 'static;

    fn call(self, builder: OwnedMutexGuard<PassableState>) -> Self::Future;
}
```

Code listing 39 The Handler trait

Then it is just required to implement the `call` function for all possible parameters. The `call` function for 2 parameters is defined in Code listing 40.

```
 1 impl<F, A, B, Fut> Handler<(A, B)> for F
 2 where
 3     F: FnOnce(A, B) -> Fut + Clone + Send + 'static,
 4     Fut: Future<Output = ()> + Send + 'static,
 5     A: FromPassable + Debug,
 6     B: FromPassable + Debug,
 7 {
 8     type Future = Pin<Box<dyn Future<Output = ()> + Send>>;
 9     fn call(self, fuzz_test_builder: OwnedMutexGuard<PassableState>)
10         -> Self::Future {
11         let a = A::from_passable(&fuzz_test_builder);
12         let b = B::from_passable(&fuzz_test_builder);
13         (self)(a, b).boxed()
14     }
15 }
```

Code listing 40 The call function for the Handler trait

The variadic parameters that can be passed to the function upon calling must implement the `FromPassable` trait, as seen in lines 5 and 6. On line 14, the function is called with parameters returned by the `from_passable` function implemented for each struct that needs to be passed.

Then the library needs the function for each possible number of parameters. In Rust, it is possible to write a macro that automatically generates all the code for the different number of parameters. The macro is defined in Code listing 41.

```
 1 macro_rules! generate_handler {
 2     ($( $($arg:ident)* ),+) => (
 3         $(
 4             impl<F, Fut, $($arg),*> Handler<($($arg),*)> for F
 5             where F: FnOnce($($arg),*) -> Fut + Clone + Send + 'static,
 6                 Fut: Future<Output = ()> + Send + 'static,
 7                 $( $arg: FromPassable + Debug ),*
 8             {
 9                 type Future = Pin<Box<dyn Future<Output = ()> + Send>>;
10                 fn call(
11                     self,
12                     fuzz_test_builder: OwnedMutexGuard<PassableState>
13                 ) -> Self::Future {
14                     $( let $arg = $arg::from_passable(&fuzz_test_builder) );*;
15                     (self)($($arg),*).boxed()
16                 }
17             }
18         )+
19     )
20 }
```

Code listing 41 The macro for generating Handler traits with variable parameters

5.4.4 State

When defining the state using the `with_state` method in the fuzz builder automatically wraps it in a `State(passed_state)` struct. That is why the developer needs to destructure the state in functions where the state is used. For example, the destructuring process is shown on line 3 in Code listing 36.

The wrap is needed for two reasons. First, the parameters injected in the developer-defined functions must implement the `FromPassable` trait. When wrapping the state in the `State` struct, the `FromPassable` trait is implemented for it by the library.

The second reason is that the state is wrapped in a `Mutex` to make it thread-safe. The `Mutex` is needed because the fuzzing process is multi-threaded. The `Mutex` is wrapped in a `Arc` to always reference the same state in the same thread.

The passed data from the developer is serialized, and when the `from_passable` function is called, the data is locked so the developer can use it. The implementation of the `from_passable` for the `State` struct is shown in Code listing 42.

```rust
#[derive(Debug)]
pub struct State<T: 'static + Send + CloneAny + Sync + Clone + Debug>
    (pub OwnedMutexGuard<T>);

impl<T: 'static + Send + CloneAny + Sync + Clone + Debug> FromPassable
    for State<T> {
    fn from_passable(builder: &OwnedMutexGuard<PassableState>) -> State<T> {
        let state = builder.state.get::<CustomArcMutex<T>>().unwrap();

        let owned_lock = task::block_in_place(move || {
            Handle::current().block_on(async move {
                state.clone_arc().lock_owned().await
            })
        });

        State(owned_lock)
    }
}
```

Code listing 42 Implementation of State struct

5.4.5 Tracing

When fuzz test fails, the developer needs access to the data that caused the failure. The fuzz tester uses the tracing library for structured logging. The logging includes information about the current sequence.

Upon failing, the library will catch the panic and log the current sequence.

5.5 CLI improvements

Trdelnik uses the Rust testing framework under the hood. With a new version of Rust, the testing framework stopped running only tests in the current folder. The ClI was fixed, and now the `trdelnik test` command runs only the tests in `trdelnik-tests` folder.

Rust testing framework also runs tests in parallel, one crate at the time. Splitting tests into multiple files makes the tests run sequentially. Instead, there is a different test runner called Nextest that can run all tests from different crates in parallel [97]. The `trdelnik test --nextest` parameter was added to the CLI to run the tests using Nextest.

There were two new commands added to the CLI. The `trdelnik fuzz new $test_name` creates a new fuzz test skeleton, and `trdelnik fuzz run $test_name`, which runs the fuzz test. The tests are stored in the `trdelnik-tests` folder.

5.6 Documentation

The documentation was just a simple README.md file in the repository's root. It contained some basic examples, but as the library grew, it was necessary to go for a more structured approach.

Rust Foundation provides a project called mdBook [98] that can be used to generate a static website from markdown files. MdBook format is well known to Rust developers as the Rust book itself is written in it. [99]

The current markdown file was structured, and the documentation was automatically generated using Github actions. The documentation is stored in the repository in the `docs` folder. The automatic build deploys it to GitHub Pages. A URL to the documentation is available in the repository's description.

Chapter 6

Evaluation of the tool

This chapter will show an example of using Trdelnik for fuzz testing and integration testing of smart contracts. The smart contract will be explained, and the test cases will be shown.

The new parallelized version using Nextest will be compared with the older version and Anchor testing framework. For all cases, there will be time for running the whole test suite and an average time for running individual tests.

All benchmarks run on Macbook Apple M1 Pro with 32GB of RAM. All benchmarks will have ten separate runs, and the average time will be the resulting value. Testing frameworks were given 20 threads to run the tests on.

The code for rerunning the benchmarks is available in the appendix A.

6.1 turnstile

The turnstile is just a simple state machine often used as an example for property-based testing.

The state machine consists of two states: locked and unlocked. The state machine has two actions: coin and pass. The state machine starts in a locked state. The coin action transitions to the unlocked state when the state machine is locked. The pass action transitions to the locked state when the state machine is unlocked. The coin action has no effect when the state machine is unlocked. The pass action has no effect when the state machine is locked.

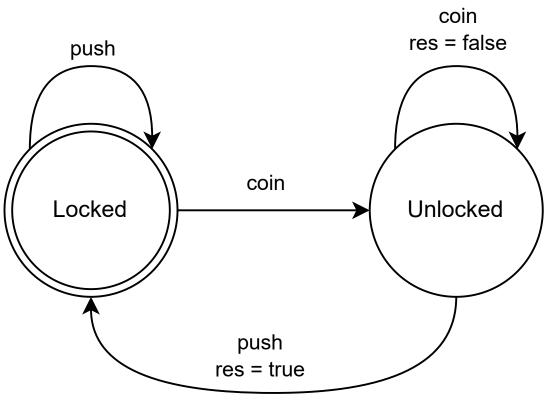


Figure 6.1 Turnstile state machine

Smart contract implementation of the turnstile is very similar to the state machine. First, the client provides an account where the state of the turnstile will be initialized. Then the client can call the coin or pass action passing this state account with the transaction. An additional flag res is set when going from unlocked to locked (pass action).

```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(())
    }
}
```

Code listing 43 Implementation of the turnstile smart contract

The implementation of flows and invariants is shown in Code listing 44. They mimic the state machine described above and the smart contract code. However, there is no need to mutate the state of the blockchain.

```rust
#[derive(Clone, Debug)]
struct TurnstileExpectedState {
    account_state: CloneableKeypair,
    locked: bool,
    res: bool,
}
async fn flow_push(
    client: Client,
    State(mut turnstile_exp_state): State<TurnstileExpectedState>
) {
    turnstile_instruction::push(
        &client,
        turnstile_exp_state.account_state.pubkey(),
        None
    ).await.expect("push failed");
    if turnstile_exp_state.locked {
        turnstile_exp_state.res = false;
    } else {
        turnstile_exp_state.locked = true;
        turnstile_exp_state.res = true;
    }
}
async fn flow_coin(
    client: Client,
    State(mut turnstile_exp_state): State<TurnstileExpectedState>
) {
    turnstile_instruction::coin(
        &client,
        "dummy_string".to_owned(),
        turnstile_exp_state.account_state.pubkey(),
        None,
    ).await.expect("coin failed");
    // Synchronize local state
    turnstile_exp_state.locked = false;
}
async fn invariant(
    client: Client,
    State(turnstile_exp_state): State<TurnstileExpectedState>
) {
    let state: AccountState = client
        .account_data(turnstile_exp_state.account_state.pubkey())
        .await
        .expect("get account data failed");
    // after pushing the turnstile should be locked
    assert_eq!(state.locked, turnstile_exp_state.locked);
    // the last push was successful
    assert_eq!(state.res, turnstile_exp_state.res);
}
```

Code listing 44 Definition of flows and invariants

Having all flows and invariants defined, we can start the fuzz testing. The fuzz testing started at line 26 in Code listing 45.

```
1
2  fn initialize_validator() -> Validator {
3      let mut validator = Validator::default();
4      validator.add_program("turnstile", PROGRAM_ID);
5      validator
6  }
7
8  async fn init_handler(
9      client: Client,
10     State(turnstile_exp_state): State<TurnstileExpectedState>
11 ) {
12     // init instruction call
13     turnstile_instruction::initialize(
14         &client,
15         turnstile_exp_state.account_state.pubkey(),
16         client.payer().pubkey(),
17         System::id(),
18         Some(turnstile_exp_state.account_state.insecure_clone()),
19     )
20     .await
21     .expect("init failed");
22 }
23
24 #[trdelnik_fuzz]
25 async fn main() {
26     FuzzTestBuilder::new()
27         .initialize_validator(initialize_validator)
28         .add_init_handler(init_handler)
29         .add_flow(flow_push)
30         .add_flow(flow_coin)
31         .with_state(TurnstileExpectedState {
32             account_state: CloneableKeypair(Keypair::new()),
33             locked: true,
34             res: false,
35         })
36         .add_invariant(invariant)
37         .start(1, 200)
38         .await;
39 }
```

Code listing 45 Turnstile fuzz test start

6.1.1 Benchmarks

When running integration tests for the turnstile, the older implementation of tests is faster for both the whole test suite and single tests 6.1. The cause could be starting the threads for the parallel implementation and joining them after the test. This overhead is not present in the serial implementation.

	average time for test suite [s]	average time for single test [s]
Trdelnik serial	10.044	5.033s
Trdelnik parallel	13.012	13.011s

Table 6.1 Benchmark for turnstile tests

6.2 d21

Explaining the framework's functions on straightforward smart contracts is suitable for understanding. However, smart contracts mimicking state machines' functionality are harder to find defects. This section will show how the framework can be used to find defects in a more complex smart contract.

D21 is a method for voting where each voter has two positive votes and a negative vote. Each vote can be used to vote for different candidates. The positive votes increase the candidate's voting count by one, while the negative vote decreases the voting count by one. The voter does not need to use all his votes [100].

The smart contract tested in this section has three parties, the election owner, candidate subjects, and voters.

6.2.1 Requirements

The smart contract has the following requirements:

- The election owner can initialize the smart contract during deployment with the duration of the election in days.
- Everyone can register a new candidate subject for the election.
- Everyone should be able to get a list of all candidates.
- Everyone can see the results of the subject.
- Only the election owner can add candidate subjects and voters.
- Every voter has two positive votes and one negative vote.
- Voter has to use positive votes before using the negative vote.
- Voter can only use votes on separate candidate subjects
- Voting ends after a specified time during initialization. After that state of the election cannot be changed.

6.2.2 Smart contract

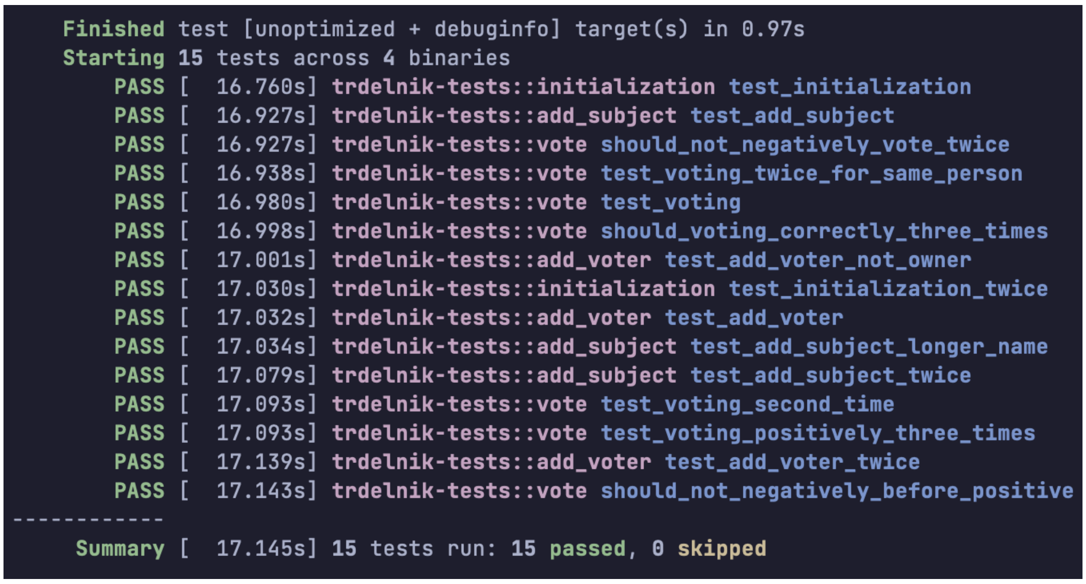
The smart contract has the following instructions:

- **initialize** - initializes the smart contract. The caller of the instruction becomes the election owner. The instruction takes the duration of the election in days as a parameter.
- **add_subject** - takes a name as a parameter and adds the transaction sender's public key as a candidate subject. The name must be less than 64 characters.
- **add_voter** - takes a voter's public key and adds it to the list of allowed voters.
- **vote** - takes a public key of a candidate subject and a vote type as parameters. The vote type can be positive or negative.

  The following accounts are used for the smart contract state storage:

- **Subject Accounts** - PDA accounts derived from the subject public key. The account stores the subject's name and the number of positive and negative votes.
- **Voter Accounts** - PDA accounts derived from the voter public key. The account stores the first vote public key, the second vote public key, and the if the user gave a negative vote
- **Basic Info Account** - Stores the election owner's public key and the end date of the election.

6.2.3 Integration tests



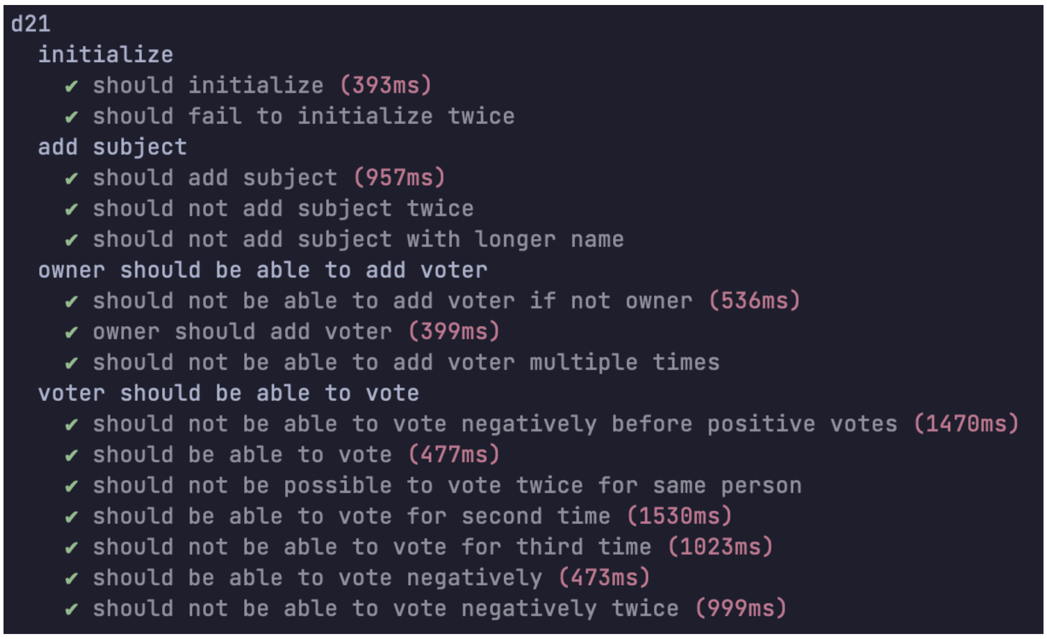
```
    Finished test [unoptimized + debuginfo] target(s) in 0.97s
    Starting 15 tests across 4 binaries
        PASS [  16.760s] trdelnik-tests::initialization test_initialization
        PASS [  16.927s] trdelnik-tests::add_subject test_add_subject
        PASS [  16.927s] trdelnik-tests::vote should_not_negatively_vote_twice
        PASS [  16.938s] trdelnik-tests::vote test_voting_twice_for_same_person
        PASS [  16.980s] trdelnik-tests::vote test_voting
        PASS [  16.998s] trdelnik-tests::vote should_voting_correctly_three_times
        PASS [  17.001s] trdelnik-tests::add_voter test_add_voter_not_owner
        PASS [  17.030s] trdelnik-tests::initialization test_initialization_twice
        PASS [  17.032s] trdelnik-tests::add_voter test_add_voter
        PASS [  17.034s] trdelnik-tests::add_subject test_add_subject_longer_name
        PASS [  17.079s] trdelnik-tests::add_subject test_add_subject_twice
        PASS [  17.093s] trdelnik-tests::vote test_voting_second_time
        PASS [  17.093s] trdelnik-tests::vote test_voting_positively_three_times
        PASS [  17.139s] trdelnik-tests::add_voter test_add_voter_twice
        PASS [  17.143s] trdelnik-tests::vote should_not_negatively_before_positive
------------
     Summary [  17.145s] 15 tests run: 15 passed, 0 skipped
```

Figure 6.2 Trdelnik CLI output

Fifteen tests were written in the older Trdelnik, the new Trdelnik with parallel execution and Anchor framework. The tests are trying to cover different situations that can occur during the election. The test cases are divided into four groups.

The first group tests the initialization and possible reinitialization of the smart contract. The test cases are *should initialize* and *should fail to initialize twice.*

The second group tests the addition of candidate subjects. The test cases are *should add subject*, *should not add subject twice*, *should not add subject with longer name than 64 chars.*



```
d21
  initialize
    ✔ should initialize (393ms)
    ✔ should fail to initialize twice
  add subject
    ✔ should add subject (957ms)
    ✔ should not add subject twice
    ✔ should not add subject with longer name
  owner should be able to add voter
    ✔ should not be able to add voter if not owner (536ms)
    ✔ owner should add voter (399ms)
    ✔ should not be able to add voter multiple times
  voter should be able to vote
    ✔ should not be able to vote negatively before positive votes (1470ms)
    ✔ should be able to vote (477ms)
    ✔ should not be possible to vote twice for same person
    ✔ should be able to vote for second time (1530ms)
    ✔ should not be able to vote for third time (1023ms)
    ✔ should be able to vote negatively (473ms)
    ✔ should not be able to vote negatively twice (999ms)
```

Figure 6.3 Anchor CLI output

The third group tests the addition of voters by the owner. The test cases are *should not be able to add voter if not owner*, *owner should add voter*, and *should not be able to add voter multiple times*.

The last group is testing the voting process. The test cases are *should not be able to vote negatively before positive votes*, *should be able to vote*, *should not be possible to vote twice for*, *should be able to vote for second time*, *should not be able to vote for third time*, *should be able to vote negatively*, *should not be able to vote negatively twice*.

The Command Line Interface tools have different outputs. The older Trdelnik output would not fit onto a page. Therefore only the revised interface and the Anchor output are shown in the following listing. The Trdelnik output is shown in Figure 6.2 and the Anchor output in Figure 6.3.

6.2.4 Benchmarks

Table 6.2 shows the results of the benchmark running the 15 tests. That said, developing the tests on Anchor was more challenging as the tests are dependent, and changing one test could result in two failing ones. That is why the average time for a single test in Anchor is less than a second because it does not need to wait until the node starts.

	average time for test suite [s]	average time for single test [s]
Anchor single node tests	11.862	0.8519
Trdelnik serial	126.14	11
Trdelnik parallel	17.145	16.998

Table 6.2 Benchmark for D21 tests

The speed up for Trdelnik running in parallel is significant - over seven times faster than the serial version

6.2.5 Fuzz tests

Fuzz tests were written for the smart contract as well. The fuzz tests primarily test the voting process as it is the most complex part of the smart contract. Having the correct votes is crucial for the smart contract to work correctly.

The overview of used flows and invariants are in Code listing 46. There are two states `SubjectState` and `VoterState`. The `SubjectState` stores the subjects and their votes. The `VoterState` stores the voters, whom they voted for, and the election owner.

```rust
struct VotingInfo {
    voter: CloneableKeypair,
    voter_acc: Pubkey,
    first_positive_vote: Option<Pubkey>,
    second_positive_vote: Option<Pubkey>,
    third_negative_vote: Option<Pubkey>,
}
struct SubjectInfo {
    subject: Pubkey,
    subject_acc: Pubkey,
    name: String,
    votes: i128,
}
struct SubjectState {
    subjects: Vec<SubjectInfo>,
}
struct VoterState {
    owner: Option<CloneableKeypair>,
    voters: Vec<VotingInfo>,
}
#[trdelnik_fuzz]
async fn main() {
    FuzzTestBuilder::new()
        .initialize_validator(initialize_validator)
        .with_state(SubjectState { subjects: vec![] })
        .with_state(VoterState { owner: None, voters: vec![] })
        .add_init_handler(init_contract)
        .add_invariant(invariant)
        .add_flow(flow_add_subject)
        .add_flow(flow_add_voter)
        .add_flow(flow_vote)
        .start(10, 200)
}
```

Code listing 46 Overview of the Fuzz test for D21

The `flow_add_subject` and `flow_add_voter` are used to add subjects and voters to the smart contract. The most interesting part is the `flow_vote`

The `flow_vote` is the most complex part of the fuzz test. It gets a random voter. For that voter it gets a random subject that the voter did not vote for yet, as seen in lines 8-10 in Code listing 47. Based on the number of votes the voter has, it calls the vote function for the subject.

```rust
async fn flow_vote(
    client: Client,
    State(mut test_state): State<SubjectState>,
    State(mut voter_state): State<VoterState>,
) {
    let voting_info = match random_voter(&mut voter_state) {
        Some(voting_info) => voting_info,
        None => return,
    };

    let filtered_subject = random_subject(&mut test_state.subjects, |subject| {
        Some(subject.subject) != voting_info.first_positive_vote
            && Some(subject.subject) != voting_info.second_positive_vote
            && Some(subject.subject) != voting_info.third_negative_vote
    });
    let subject_info = match filtered_subject {
        Some(subject) => subject,
        None => return,
    };
    match voting_info {
        VotingInfo {
            first_positive_vote: None,
            ..
        } => {
            call_vote(voting_info, &client, true, subject_info).await;
            subject_info.votes += 1;
            voting_info.first_positive_vote = Some(subject_info.subject);
        }
        VotingInfo {
            second_positive_vote: None,
            ..
        } => {
            call_vote(voting_info, &client, true, subject_info).await;
            subject_info.votes += 1;
            voting_info.second_positive_vote = Some(subject_info.subject);
        }
        VotingInfo {
            third_negative_vote: None,
            ..
        } => {
            call_vote(voting_info, &client, false, subject_info).await;
            subject_info.votes -= 1;
            voting_info.third_negative_vote = Some(subject_info.subject);
        }
        _ => {}
    }
}
```

Code listing 47 Vote flow for D21

After the vote is cast, the invariant is checked. The invariant checks that the number of votes for each subject is correct. The invariant is shown in Code listing 48. The invariant is called after each flow, so it is checked after each vote.

```rust
async fn invariant(client: Client, State(test_state): State<SubjectState>) {
    let program = client.program(PROGRAM_ID);
    let subjects = program
        .accounts::<SubjectAccount>(vec![])
        .expect("Unable to get subjects");
    assert!(subjects.len() == test_state.subjects.len());
    for subject in test_state.subjects.iter() {
        let subject_acc = program
            .account::<SubjectAccount>(subject.subject_acc)
            .expect("Unable to get subject");
        assert!(subject_acc.name == subject.name);
        assert!(subject_acc.votes as i128 == subject.votes);
    }
}
```

■ **Code listing 48** Invariant for D21

Fuzz tests found a bug in the smart contract when initializing with a random number of days. Solana returns a date in a signed integer of 64 bits. The days get passed an unsigned integer of 32 bits.

Following code `clock.unix_timestamp + (election_duration_days * DAY_IN_SECONDS)` transfers the days to seconds, which means it can overflow the i64. The bug was fixed by transferring the Solana unix timestamp to u128 and limiting the number of days.

Chapter 7

Conclusion and future work

This thesis aimed to compare Rust blockchain networks in terms of their architecture, state of developer and security tooling, and other relevant aspects. The developer tooling was compared in terms of the documentation, smart contract language used, and technology-specific additions to the language. The security tooling was compared in terms of static analysis tools, the existence of unit and integration testing frameworks, and fuzzing frameworks.

After knowing the state of these blockchain technologies, Solana was chosen as a blockchain to improve its ecosystem. The testing framework Trdelnik was improved with parallelized runtime for tests with automatic start of test node validators. Also, there were improvements for CLI output, the code generation tool, and the documentation.

The improved integration testing framework with parallelized runtime was benchmarked on smart contract with fifteen tests and showed over seven times shorter test suite processing time. Benchmarks supporting this claim can be found at 6.2.

The fuzz testing framework was not included in any blockchain technologies mentioned in the analysis chapter of this thesis. Solana with Trdelnik is now the only blockchain technology with a fuzz testing framework.

Even though the fuzz testing framework runs the fuzzing in parallel on multiple test blockchain nodes, the developer does not need to handle access to memory or synchronization of the fuzzing process. The API is designed for developer convenience and allows the developer to assign custom data structures that get automatically injected into the parameters of their handlers.

The fuzz testing framework was used on multiple smart contracts, including turnstile, anchor-counter, and D21. On the D21 smart contract, the fuzz testing framework found a bug in the smart contract that was not found when the integration tests were written.

The fuzz testing framework should also provide a random data generator instance. That would be beneficial in case of an error. To reproduce the error, the developer could rerun the fuzzing process with the same random data generator instance.

Although the enhancements to the testing framework are noteworthy, there is still room for further refinement. When the fuzzing fails, the developer must go through the logs to find the cause of the error. The fuzz testing framework could be improved with the option to export logs from the fuzzing process.

After the console for Trdelnik is implemented, it would be helpful if the node validator would not be stopped, as the developer could query the blockchain and internal state to find the cause of the error.

Upon starting the fuzz testing, it becomes a passive experience as the developer has to wait until a bug is found. That could be improved by having a better CLI allowing the programmer to view debug logs and individual threads of the fuzzing process.

The potential for enhancement is not limited to the fuzzing framework alone. The code generation tool could also be improved to generate common testing code, like generating PDAs for the smart contract accounts or retrieving all accounts owned by the program. The integration testing framework could also benefit from an improved code generation tool.

One feature was only tested and not implemented in the integration testing framework. The feature was the code coverage of the tested smart contract. Code coverage is a complex feature to implement as the code is not running in the same process as the tests. The smart contract code runs inside the validator in the BPF. It would be required that the test validator node would support sending back lines of code that were executed during the transaction. The BPF also receives only the compiled smart contract code. Therefore it would require a particular compilation target as well.

Appendix A
Project repositories

- The code for the implementation of the Trdelnik framework and turnstile example is available here:
  https://github.com/reastyn/trdelnik
- The code for the D21 smart contract is available here:
  https://github.com/reastyn/anchor-d21
- The code for the anchor-counter smart contract is available here:
  https://github.com/reastyn/anchor-counter
- The code gathering the data for statistics is available here:
  https://gist.github.com/reastyn/d3f71c760af68c6e3ae317da06e88d13

Bibliography

1. FERNÁNDEZ-CARAMÉS, Tiago M; FRAGA-LAMAS, Paula. A Review on the Use of Blockchain for the Internet of Things. *Ieee Access*. 2018, vol. 6, pp. 32979–33001.
2. HÖLBL, Marko; KOMPARA, Marko; KAMIŠALIĆ, Aida; NEMEC ZLATOLAS, Lili. A systematic review of the use of blockchain in healthcare. *Symmetry*. 2018, vol. 10, no. 10, p. 470.
3. TRELEAVEN, Philip; GENDAL BROWN, Richard; YANG, Danny. Blockchain Technology in Finance. *Computer*. 2017, vol. 50, no. 9, pp. 14–17. Available from DOI: `10.1109/MC.2017.3571047`.
4. HAFID, Abdelatif; HAFID, Abdelhakim Senhaji; SAMIH, Mustapha. Scaling Blockchains: A Comprehensive Survey. *IEEE Access*. 2020, vol. 8, pp. 125244–125262. Available from DOI: `10.1109/ACCESS.2020.3007251`.
5. *Small Business Retail — Visa* [online]. [N.d.]. [visited on 2023-03-30]. Available from: `https://usa.visa.com/run-your-business/small-business-tools/retail.html`.
6. LI, Wenzheng; HE, Mingsheng. Comparative Analysis of Bitcoin, Ethereum, and Libra. In: *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*. 2020, pp. 545–550. Available from DOI: `10.1109/ICSESS49938.2020.9237710`.
7. CHEN, Huashan; PENDLETON, Marcus; NJILLA, Laurent; XU, Shouhuai. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*. 2020, vol. 53, no. 3, pp. 1–43.
8. KANNENGIESSER, Niclas; LINS, Sebastian; SANDER, Christian; WINTER, Klaus; FREY, Hellmuth; SUNYAEV, Ali. Challenges and Common Solutions in Smart Contract Development. *IEEE Transactions on Software Engineering*. 2022, vol. 48, no. 11, pp. 4291–4318. Available from DOI: `10.1109/TSE.2021.3116808`.
9. NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*. 2008, p. 21260.
10. ZOHAR, Aviv. Bitcoin: under the hood. *Communications of the ACM*. 2015, vol. 58, no. 9, pp. 104–113.
11. MÖSER, Malte; EYAL, Ittay; GÜN SIRER, Emin. Bitcoin Covenants. In: CLARK, Jeremy; MEIKLEJOHN, Sarah; RYAN, Peter Y.A.; WALLACH, Dan; BRENNER, Michael; ROHLOFF, Kurt (eds.). *Financial Cryptography and Data Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 126–141. ISBN 978-3-662-53357-4.

12. BISTARELLI, Stefano; MAZZANTE, Gianmarco; MICHELETTI, Matteo; MOSTARDA, Leonardo; SESTILI, Davide; TIEZZI, Francesco. Ethereum smart contracts: Analysis and statistics of their source code and opcodes. *Internet of Things*. 2020, vol. 11, p. 100198. ISSN 2542-6605. Available from DOI: `https://doi.org/10.1016/j.iot.2020.100198`.
13. BUTERIN, Vitalik et al. A next-generation smart contract and decentralized application platform. *white paper*. 2014, vol. 3, no. 37, pp. 2–1.
14. MATSAKIS, Nicholas D.; KLOCK, Felix S. The Rust Language. In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 103–104. HILT '14. ISBN 9781450332170. Available from DOI: `10.1145/2663171.2663188`.
15. SIMONE, Sergio De. *Linux 6.1 Officially Adds Support for Rust in the Kernel* [online]. 2022. [visited on 2023-03-09]. Available from: `https://www.infoq.com/news/2022/12/linux-6-1-rust/`.
16. *Rust Foundation - Hello World!* [online]. 2021. [visited on 2023-03-08]. Available from: `https://foundation.rust-lang.org/news/2021-02-08-hello-world/`.
17. *Sir Antony Hoare — Computer History Museum* [online]. [N.d.]. [visited on 2023-02-05]. Available from: `https://web.archive.org/web/20151115012103/http://www.computerhistory.org/fellowawards/hall/bios/Antony,Hoare`.
18. HOARE, Tony. *Null References: The Billion Dollar Mistake* [online]. 2009. [visited on 2023-02-20]. Available from: `https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/`.
19. *near_bindgen — NEAR Documentation* [online]. [N.d.]. [visited on 2023-03-05]. Available from: `https://web.archive.org/web/20221209232246/https://docs.near.org/sdk/rust/contract-structure/near-bindgen`.
20. KLABNIK, Steve; NICHOLS, Carol. *The Rust programming language*. No Starch Press, 2023.
21. MIGHAN, Soosan Naderi; MIŠIĆ, Jelena; MIŠIĆ, Vojislav B. On block delivery time in Ethereum network. In: *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 2022, pp. 2867–2872. Available from DOI: `10.1109/GLOBECOM48099.2022.10001081`.
22. *GitHub - move-language/move* [online]. [N.d.]. [visited on 2023-02-28]. Available from: `https://github.com/move-language/move`.
23. *Write Smart Contracts with Sui Move — Sui Docs* [online]. 2023. [visited on 2023-03-20]. Available from: `https://docs.sui.io/build/move`.
24. *Move on Aptos* [online]. 2023. [visited on 2023-02-07]. Available from: `https://aptos.dev/guides/move-guides/move-on-aptos/`.
25. *GitHub - Zellic/move-prover-examples: A gentle, example-based guide to getting started with the Move prover.* [online]. [N.d.]. [visited on 2023-04-03]. Available from: `https://github.com/Zellic/move-prover-examples`.
26. ZHONG, Jingyi; CHEANG, Kevin; QADEER, Shaz; GRIESKAMP, Wolfgang; BLACKSHEAR, Sam; PARK, Junkil; ZOHAR, Yoni; BARRETT, Clark; DILL, David. The Move Prover. In: 2020, pp. 137–150. ISBN 978-3-030-53287-1. Available from DOI: `10.1007/978-3-030-53288-8\_7`.
27. *move-language/move* [online]. [N.d.]. [visited on 2023-02-27]. Available from: `https://github.com/move-language/move/blob/main/language/documentation/tutorial/README.md`.
28. *The Move Language - The Move Book* [online]. 2022. [visited on 2023-02-28]. Available from: `https://move-book.com`.

29. *Artemis metrics* [online]. 2023. [visited on 2023-04-27]. Available from: `https://app.artemis.xyz/dashboard`.

30. MENS, Tom; DEMEYER, Serge; ZIMMERMANN, Thomas; NAGAPPAN, Nachiappan; ZELLER, Andreas. Predicting bugs from history. *Software evolution*. 2008, pp. 69–88.

31. WALLACE, Linda G.; SHEETZ, Steven D. The adoption of software measures: A technology acceptance model TAM perspective. *Information and Management*. 2014, vol. 51, no. 2, pp. 249–259. ISSN 0378-7206. Available from DOI: `https://doi.org/10.1016/j.im.2013.12.003`.

32. KHANAM, Zeba; AHSAN, Mohammed Najeeb. Evaluating the effectiveness of test driven development: Advantages and pitfalls. *International Journal of Applied Engineering Research*. 2017, vol. 12, no. 18, pp. 7705–7716.

33. TOSUN, Ayse; AHMED, Muzamil; TURHAN, Burak; JURISTO, Natalia. On the Effectiveness of Unit Tests in Test-Driven Development. In: *Proceedings of the 2018 International Conference on Software and System Process*. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 113–122. ICSSP '18. ISBN 9781450364591. Available from DOI: `10.1145/3202710.3203153`.

34. AYEWAH, Nathaniel; PUGH, William; HOVEMEYER, David; MORGENTHALER, J. David; PENIX, John. Using Static Analysis to Find Bugs. *IEEE Software*. 2008, vol. 25, no. 5, pp. 22–29. Available from DOI: `10.1109/MS.2008.130`.

35. FEIST, Josselin; GRIECO, Gustavo; GROCE, Alex. Slither: a static analysis framework for smart contracts. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

36. ZAYOUR, Iyad; HAJJDIAB, Hassan. How much integrated development environments ides improve productivity? *J. Softw.* 2013, vol. 8, no. 10, pp. 2425–2431.

37. *Stack Overflow Developer Survey 2022* [online]. 2023. [visited on 2023-03-20]. Available from: `https://survey.stackoverflow.co/2022/`.

38. TEAM, The Rust Core. *4 years of Rust — Rust Blog* [online]. 2019. [visited on 2023-03-10]. Available from: `https://blog.rust-lang.org/2019/05/15/4-Years-Of-Rust.html`.

39. MARR, Bernard. *Blockchain: A Very Short History Of Ethereum Everyone Should Read* [online]. 2018. [visited on 2023-02-24]. Available from: `https://www.forbes.com/sites/bernardmarr/2018/02/02/blockchain-a-very-short-history-of-ethereum-everyone-should-read/?sh=72b5e6bc1e89`.

40. *About Sui — Sui Docs* [online]. 2023. [visited on 2023-02-03]. Available from: `https://docs.sui.io/devnet/learn/about-sui`.

41. WANG, Angela. *Sui Blockchain Guide: Token sale details announced!* [online]. 2023. [visited on 2023-03-22]. Available from: `https://boxmining.com/sui-blockchain/`.

42. CRYPTO.COM. *What Is Delegated Proof of Stake?* [online]. 2023. [visited on 2023-02-15]. Available from: `https://crypto.com/university/what-is-dpos-delegated-proof-of-stake`.

43. *How Sui Works — Sui Docs* [online]. 2023. [visited on 2023-02-21]. Available from: `https://docs.sui.io/learn/how-sui-works`.

44. *shared_auction.move smart contract code* [online]. 2023. [visited on 2023-02-13]. Available from: `https://github.com/MystenLabs/sui/blob/da51749786e6a2a43388e7cebd9c3ec475dc69bd/sui_programmability/examples/nfts/sources/shared_auction.move`.

45. *Objects — Sui Docs* [online]. 2023. [visited on 2023-03-21]. Available from: `https://docs.sui.io/devnet/learn/objects`.

46. *Build and Test the Sui Move Package — Sui Docs* [online]. 2023. [visited on 2023-03-23]. Available from: `https://docs.sui.io/devnet/build/move/build-test`.
47. FOUNDATION, Sui. *Announcing Enhanced Move VSCode Plugin* [online]. 2022. [visited on 2023-03-25]. Available from: `https://blog.sui.io/enhanced-move-vscode-plugin/`.
48. *GitHub - pontem-network/intellij-move: Support for Move, smart-contract language for Aptos and other blockchains* [online]. 2023. [visited on 2023-02-23]. Available from: `https://github.com/pontem-network/intellij-move`.
49. *auction.move smart contract code* [online]. 2023. [visited on 2023-02-13]. Available from: `https://github.com/MystenLabs/sui/blob/da51749786e6a2a43388e7cebd9c3ec475dc69bd/sui_programmability/examples/nfts/sources/auction.move`.
50. *sui/sdk/typescript at main, MystenLabs/sui, GitHub* [online]. 2023. [visited on 2023-03-24]. Available from: `https://github.com/MystenLabs/sui/tree/main/sdk/typescript/`.
51. *sui/sdk/wallet-adapter at main, MystenLabs/sui, GitHub* [online]. 2023. [visited on 2023-03-25]. Available from: `https://github.com/MystenLabs/sui/tree/main/sdk/wallet-adapter`.
52. *Playing Tic-Tac-Toe — Sui Docs* [online]. 2023. [visited on 2023-03-24]. Available from: `https://docs.sui.io/explore/tutorials#playing-tictactoe`.
53. *Aptos Whitepaper* [online]. 2022. [visited on 2023-02-11]. Available from: `https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf`.
54. *Aptos Blockchain Guide: the Next Big Innovation in Blockchain Scaling* [online]. 2023. [visited on 2023-02-08]. Available from: `https://boxmining.com/aptos-blockchain/`.
55. Aptos Whitepaper [online]. 2022 [visited on 2023-03-13]. Available from: `https://aptos.dev/assets/files/Aptos-Whitepaper-47099b4b907b432f81fc0effd34f3b6a.pdf`.
56. *Aptos Unit Testing* [online]. 2023. [visited on 2023-02-08]. Available from: `https://aptos.dev/guides/move-guides/book/unit-testing`.
57. *Aptos Transactional Testing* [online]. 2023. [visited on 2023-02-08]. Available from: `https://web.archive.org/web/20221005151554/https://aptos.dev/guides/move-guides/guide-move-transactional-testing`.
58. *GitHub - aptos-labs/aptos-wallet-adapter: A monorepo modular wallet adapter for Aptos applications* [online]. 2023. [visited on 2023-02-11]. Available from: `https://github.com/aptos-labs/aptos-wallet-adapter`.
59. *Aptos Wallet* [online]. 2023. [visited on 2023-03-31]. Available from: `https://web.archive.org/web/20221206170600/https://aptos.dev/guides/install-petra-wallet-extension/`.
60. [online]. 2023. [visited on 2023-02-28]. Available from: `https://playground.pontem.network/`.
61. *NEAR — The OS for an Open Web* [online]. [N.d.]. [visited on 2023-03-05]. Available from: `https://near.org/papers/the-official-near-white-paper/#why-near`.
62. *NEAR — The OS for an Open Web* [online]. [N.d.]. [visited on 2023-03-02]. Available from: `https://near.org/blog/near-mainnet-is-now-community-operated/`.
63. *Hello NEAR — NEAR Documentation* [online]. 2023. [visited on 2023-03-01]. Available from: `https://docs.near.org/tutorials/examples/hello-near`.
64. *GitHub - blocksecteam/rustle: A static analyzer for NEAR smart contract in Rust* [online]. [N.d.]. [visited on 2023-03-11]. Available from: `https://github.com/blocksecteam/rustle`.

65. YAKOVENKO, Anatoly. Solana: A new architecture for a high performance blockchain v0.8.13. [N.d.]. Available also from: https://coincode-live.github.io/static/whitepaper/source001/10608577.pdf.

66. *Solana Year in Review 2020. 2020 was a difficult and challenging — by Solana — Solana — Medium* [online]. 2021. [visited on 2023-03-15]. Available from: https://medium.com/solana-labs/year-in-review-2020-c3731d1cc8a.

67. YAKOVENKO, Anatoly. *Sealevel, Parallel Processing Thousands of Smart Contracts* [online]. [N.d.]. [visited on 2023-03-12]. Available from: https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192.

68. *Program Derived Addresses — Solana Cookbook* [online]. 2022. [visited on 2023-03-07]. Available from: https://solanacookbook.com/core-concepts/pdas.html#generating-pdas.

69. *InterfaceDescriptionLanguage - Device Description Working Group Wiki* [online]. 2008. [visited on 2023-02-22]. Available from: https://www.w3.org/2005/MWI/DDWG/wiki/InterfaceDescriptionLanguage.

70. CUI, Siwei; ZHAO, Gang; GAO, Yifei; TAVU, Tien; HUANG, Jeff. VRust: Automated Vulnerability Detection for Solana Smart Contracts. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 639–652. CCS '22. ISBN 9781450394505. Available from DOI: 10.1145/3548606.3560552.

71. SEC3. *Soteria, A vulnerability scanner for Solana smart contracts — by sec3 — Coinmonks — Medium* [online]. 2021. [visited on 2023-03-19]. Available from: https://medium.com/coinmonks/soteria-a-vulnerability-scanner-for-solana-smart-contracts-cc202cf17c99.

72. *Anchor - Visual Studio Marketplace* [online]. [N.d.]. [visited on 2023-02-04]. Available from: https://marketplace.visualstudio.com/items?itemName=Ayushh.vscode-anchor.

73. *Solana — Ecosystem* [online]. [N.d.]. [visited on 2023-03-16]. Available from: https://solana.com/ecosystem/explore?categories=wallet.

74. *Solpg.io* [online]. 2023. [visited on 2023-03-19]. Available from: https://beta.solpg.io/.

75. *SolDev* [online]. [N.d.]. [visited on 2023-03-18]. Available from: https://www.soldev.app/.

76. *Your First Move Module — Aptos Docs* [online]. 2023. [visited on 2023-02-10]. Available from: https://aptos.dev/tutorials/first-move-module/.

77. *Unit Tests — NEAR Documentation* [online]. 2023. [visited on 2023-03-03]. Available from: https://docs.near.org/sdk/rust/testing/unit-tests.

78. *Test your contracts - Truffle Suite* [online]. 2023. [visited on 2023-03-29]. Available from: https://trufflesuite.com/docs/truffle/how-to/debug-test/test-your-contracts/.

79. *Unit Testing with Pytest - Brownie v1.0.0 documentation* [online]. 2021. [visited on 2023-02-14]. Available from: https://eth-brownie.readthedocs.io/en/v1.0.0%5C_a/tests.html.

80. *Ganache — Overview - Truffle Suite* [online]. 2023. [visited on 2023-02-18]. Available from: https://trufflesuite.com/docs/ganache/.

81. *Forking other networks — Ethereum development environment for professionals by Nomic Foundation* [online]. [N.d.]. [visited on 2023-02-19]. Available from: https://hardhat.org/hardhat-network/docs/guides/forking-other-networks.

82. *GitHub - crytic/echidna: Ethereum smart contract fuzzer* [online]. [N.d.]. [visited on 2023-02-17]. Available from: https://github.com/crytic/echidna.

83. *Fuzzing - Woke* [online]. 2023. [visited on 2023-04-02]. Available from: `https://ackeeblockchain.com/woke/docs/latest/testing-framework/fuzzing/`.

84. *Tools for Solidity - Visual Studio Marketplace* [online]. [N.d.]. [visited on 2023-03-27]. Available from: `https://marketplace.visualstudio.com/items?itemName=AckeeBlockchain.tools-for-solidity`.

85. *Solidity - IntelliJ IDEs Plugin — Marketplace* [online]. [N.d.]. [visited on 2023-02-23]. Available from: `https://plugins.jetbrains.com/plugin/9475-solidity`.

86. *GitHub - crytic/slither: Static Analyzer for Solidity* [online]. [N.d.]. [visited on 2023-03-13]. Available from: `https://github.com/crytic/slither`.

87. *GitHub - near/wallet-selector: This is a wallet selector modal that allows users to interact with NEAR dApps with a selection of available wallets.* [online]. [N.d.]. [visited on 2023-03-04]. Available from: `https://github.com/near/wallet-selector`.

88. *sui/sdk/wallet-adapter at main, MystenLabs/sui, GitHub* [online]. [N.d.]. [visited on 2023-03-26]. Available from: `https://github.com/MystenLabs/sui/tree/main/sdk/wallet-adapter`.

89. *GitHub - solana-labs/wallet-adapter: Modular TypeScript wallet adapters and components for Solana applications.* [online]. [N.d.]. [visited on 2023-03-17]. Available from: `https://github.com/solana-labs/wallet-adapter`.

90. KLEES, George; RUEF, Andrew; COOPER, Benji; WEI, Shiyi; HICKS, Michael. Evaluating Fuzz Testing. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. CCS '18. ISBN 9781450356930. Available from DOI: `10.1145/3243734.3243804`.

91. GODEFROID, Patrice. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. In: *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. Atlanta, Georgia: Association for Computing Machinery, 2007, p. 1. RT '07. ISBN 9781595938817. Available from DOI: `10.1145/1292414.1292416`.

92. SEREBRYANY, Kostya. OSS-Fuzz - Google's continuous fuzzing service for open source software. In: Vancouver, BC: USENIX Association, 2017.

93. GROCE, Alex; GRIECO, Gustavo. Echidna-Parade: A Tool for Diverse Multicore Smart Contract Fuzzing. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual, Denmark: Association for Computing Machinery, 2021, pp. 658–661. ISSTA 2021. ISBN 9781450384599. Available from DOI: `10.1145/3460319.3469076`.

94. *solana/validator at master, solana-labs/solana, GitHub* [online]. [N.d.]. [visited on 2023-03-15]. Available from: `https://github.com/solana-labs/solana/tree/master/validator`.

95. *Traits: Defining Shared Behavior - The Rust Programming Language* [online]. 2023. [visited on 2023-03-28]. Available from: `https://doc.rust-lang.org/book/ch10-02-traits.html`.

96. WILSON, James. *jsdw - Rust: Fun with Function Traits* [online]. 2020. [visited on 2023-02-18]. Available from: `https://jsdw.me/posts/rust-fn-traits/`.

97. *How nextest works - cargo-nextest* [online]. 2023. [visited on 2023-03-06]. Available from: `https://nexte.st/book/how-it-works.html`.

98. *Introduction - mdBook Documentation* [online]. 2023. [visited on 2023-02-25]. Available from: `https://rust-lang.github.io/mdBook/`.

99. *The Rust Programming Language - The Rust Programming Language* [online]. 2023. [visited on 2023-03-10]. Available from: `https://doc.rust-lang.org/book/`.

100. JANEČEK, Karel. *D21 - Janeckova metoda* [online]. 2022. [visited on 2023-02-16]. Available from: `https://www.ih21.org/en/d21-janecek-method`.

Contents of attached archive file

`readme.txt`	file with information about the contents
`src`	source codes
`src/trdelnik`	Trdelnik framework with examples like turnstile
`src/d21`	D21 smart contract with integrational and fuzzer tests
`src/anchor-counter`	anchor-counter smart contract with integrational tests
`src/data-scraper.py`	script for gathering data for comparison table
`thesis`	the thesis in LaTeX
`thesis/thesis.pdf`	the thesis in PDF format