Ministry of Education and Science of Ukraine

National University of "Kyiv-Mohyla Academy"

Department of Informatics of the Faculty of Informatics

Creation of an atomic cross-chain escrow, which supports EVM, Solana and TON

Text part to the thesis in the specialty "Software engineereing" - 121

Thesis Supervisor:

Seniour lecturer

K.S. Gorokhovskyi

_____ (Signature)

"___" _____ 2025

Done by Student:

V.O. Rastiehaiev

"___" _____ 2025

Kyiv, 2025

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"
Department of Informatics of the Faculty of Informatics

Approved
Head of Department of Informatics,
Associate Professor, Ph.D.
S. S. Gorokhovskyi

_____ (Signature)

"___" _____ 2025

INDIVIDUAL TASK for thesis
of the student V.O. Rastiehaiev
4th year of the Faculty of Informatics
topic: Creation of an atomic cross-chain escrow, which supports EVM, Solana and TON

The content of the PM to the thesis:
Abstract
Introduction
Theoretical background
Related work
Methodology
Evaluation
Conclusions
Bibliography

Date of issue: "___" _____ 2025

Supervisor: _____ (Signature)

Task received: _____ (Signature)

Abstract

This paper presents the design, development, and evaluation of an atomic cross-chain escrow system facilitating secure, trustless asset transfers between Ethereum Virtual Machine compatible blockchains, Solana, and The Open Network. Driven by the increasing fragmentation of the blockchain ecosystem and the demand for seamless cross-chain interoperability, this work addresses the limitations of centralized intermediaries and existing bridge solutions. The research involved identifying key challenges in atomic cross-chain escrow development, including differing blockchain architectures and HTLC concept and its core principles. As a core contribution, three distinct smart contracts were engineered, one for each target blockchain (EVM, Solana, and TON), to enable a hash-time-locked escrow mechanism. The developed contracts were evaluated for security and efficiency, demonstrating the feasibility of achieving guaranteed fund delivery or refund across these disparate networks without reliance on a trusted third party.

Contents

Chapter 1

Introduction

1.1 Background

The dawn of Bitcoin in 2009 marked the beginning of a technological revolution: blockchain. Initially conceived as the underlying technology for a peer-to-peer electronic cash system, blockchain's core principles: decentralization, transparency, and immutability - quickly demonstrated potential far beyond digital currencies. During the past decade, this potential has caused growth in popularity and interest, transforming blockchain from a niche concept into a major field attracting significant attention from developers, entrepreneurs, investors, and established industries.

The landscape is no longer dominated by a single entity. Instead, we witness a vibrant and rapidly expanding ecosystem comprising a large number of distinct blockchains. Each often possesses unique technical characteristics, consensus mechanisms, governance models, and intended use cases, ranging from general-purpose smart contract platforms like Ethereum and Solana to specialized chains focused on finance, digital identity, or gaming. The sheer volume of digital assets managed across these different networks represents substantial economic value, running into billions or even trillions of dollars, underscoring the high stakes involved in their secure and efficient operation.

1.2 Motivation

While the diversity of blockchains facilitates innovation and specialization, it simultaneously creates complexities. The current reality is one of digital "islands": sealed networks largely unable to communicate or exchange transactions directly with one another. This lack of inherent interoperability presents a major bottleneck for users and developers. Consider a user holding valuable tokens on Blockchain A who wishes to utilize a decentralized application exclusively available on Blockchain B, or perhaps wants to exchange those tokens for a different asset native to Blockchain C.

Currently, performing such cross-chain swaps or interactions is often a complex, costly, and potentially insecure process. Users typically rely on centralized exchanges (which introduce third-party risk and negate some benefits of decentralization) or intricate, multi-step bridging solutions that can be technically challenging and vulnerable to exploits. Therefore, this restricts the seamless flow of value and data across blockchains, limits the potential of decentralized applications, and worsens the quality of user experience.

Addressing this challenge is crucial for the continued growth and maturation of the blockchain ecosystem. Enabling secure, efficient, and trustless interaction between different blockchains is not only desirable, but necessary to achieve true interoperability.

1.3 Objectives and Scope

This paper directly addresses interoperability issue. By exploring and developing mechanisms for HTLC-based atomic cross-chain escrow, the aim is to provide a foundational component to enable users to swap assets or engage in transactions across selected blockchain networks reliably and without unnecessary intermediaries. The atomic nature of this system ensures that a cross-chain exchange either completes successfully on all participating chains or fails entirely, returning all assets to their original owners. This all-or-nothing proposition eliminates the significant risks associated with partially executed transactions and the potential for funds becoming permanently locked or lost due to failures in complex bridging or intermediary systems.

The work specifically targets three diverse and influential blockchain architectures: EVM-compatible networks, Solana, and The Open Network. By focusing on these distinct platforms, this work aims to demonstrate the adaptability of the atomic escrow concept across different blockchain concepts and smart contract models.

Chapter 2

Theoretical background

2.1 Hashed Timelock Contract

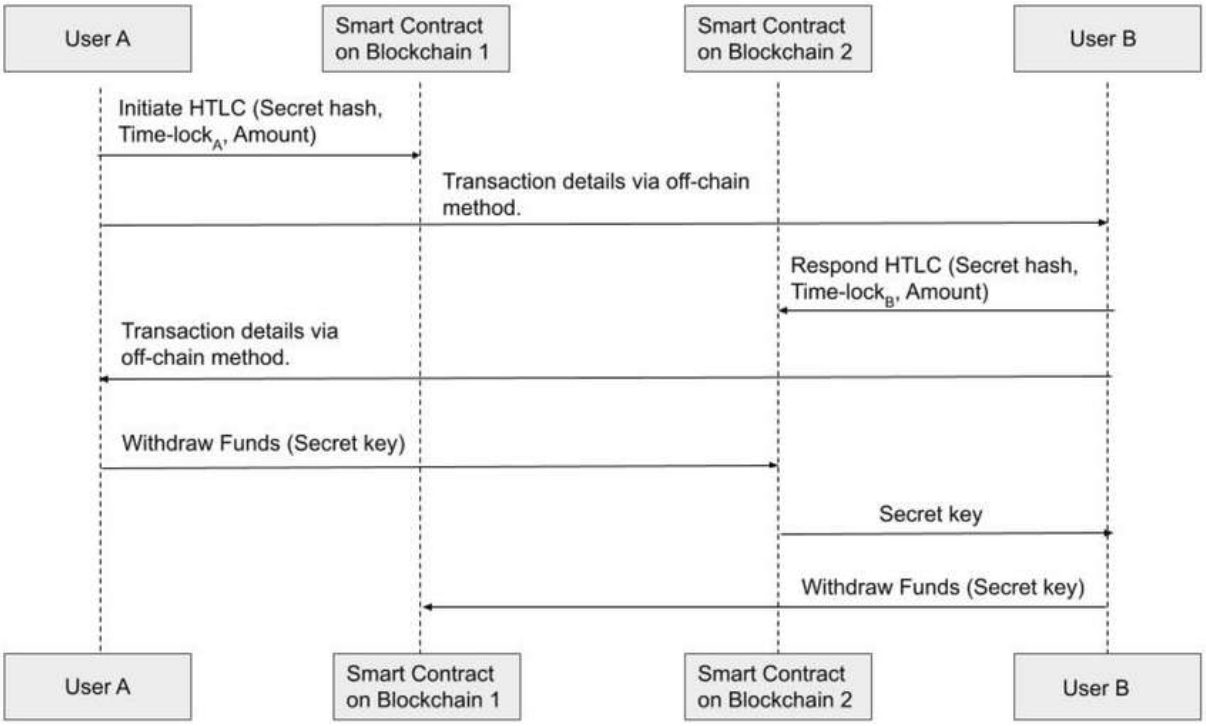

Figure 2.1: HTLC sequence diagram *Source:* Reproduced from [1], p. 37.

Hashed Timelock Contracts[1] represent a pivotal transactional agreement within the cryptocurrency industry, designed to facilitate conditional payments. These contracts enable secure and trustless transactions across disparate blockchain networks or protocols, effectively removing the need for reliance on intermediaries. The fundamental challenge HTLCs address is the inherent counterparty risk present in decentralized environments. By design, HTLCs ensure that all participating parties in a transaction either fulfill their respective obligations or the transaction is entirely voided. The architecture of HTLC is defined by two core mechanisms: the hashlock and the timelock, as seen in Figure

2.1. Together, these components create a secure payment system. HTLCs have become foundational in various critical applications within the cryptocurrency ecosystem and in the implementation of atomic swaps for trustless cross-chain exchange.

2.1.1 Core Concepts and Mechanisms of HTLC

Hashlock

One of the two core concepts in HTLC is a hashlock: a mechanism that restricts the ability to spend funds until a specific piece of data, known as the preimage, is publicly revealed. The process begins with the sender of a payment generating a secret value, the preimage. This secret is then used as an input for a cryptographic hash function, resulting in a unique output called the hashlock. The funds intended for the recipient are then locked using this hashlock. To gain access to these locked funds, the intended recipient must provide the original preimage, which when hashed using the same function, produces the hashlock. It is crucial to understand that cryptographic hash functions are designed to be one-way, meaning that it is computationally infeasible to reverse the process and derive the preimage from the hash. A significant characteristic of hashlocks is that once a preimage is revealed, it can unlock any other hashlock that was created using the same initial preimage. The use of robust cryptographic hash functions, such as SHA-256 or Keccak-256, empowers the security of the hashlock mechanism in HTLCs.

Timelock

Complementing the hashlock mechanism is the timelock, which introduces a temporal constraint on the spending of funds. A timelock restricts the ability to spend the locked cryptocurrency until a specific point in the future, measured in terms of time or the number of blocks added to the blockchain. Within the context of HTLCs, two primary types of timelocks exist: absolute timelocks and relative timelocks. Absolute timelocks specify a particular time or block height at which the funds can be accessed. Once this predetermined time or block height is reached, the funds become available to the recipient. Relative timelocks operate based on the number of blocks that have been generated since a prior transaction involving the funds was confirmed. These timelocks ensure that a certain amount of time, measured in blocks, must pass before the transaction can be finalized. The fundamental purpose of timelocks in HTLCs is to ensure that transactions are rejected until the specified temporal conditions are met. Furthermore, timelocks play a crucial role in providing a mechanism for the sender to receive a refund of the locked funds if the recipient fails to provide the correct preimage within the agreed-upon timeframe. The time constraints imposed by timelocks are typically hardcoded into the contract, providing a level of immutability and security, preventing the sender from altering the conditions after the HTLC has been established.

2.1.2 HTLC transaction sequence

The HTLC transaction sequence starts with the party intending to make a conditional payment, referred to as the sender (Alice) generates the preimage and obtains hashlock from it. Then Alice shares the hashlock with the intended recipient of the payment, referred to as the receiver (Bob), but the original preimage is not disclosed at this stage. Bob cannot access the preimage prematurely. This prevents scenarios, particularly in applications like atomic swaps, where Bob might be able to claim the funds before Alice has fulfilled her part of the agreement.

If Bob, the intended recipient, wishes to access the funds locked within the HTLC, he must provide the correct preimage to the smart contract before the predetermined timelock period expires. Upon receiving the preimage, the HTLC executes a verification process, comparing the hash of the provided preimage with the original hashlock that was encoded in the contract. If these two values match, confirming that the correct preimage has been presented, the HTLC then releases the locked cryptocurrency to Bob. The revelation of the preimage by Bob on one blockchain can be observed by Alice on a different blockchain. This observation then allows Alice to proceed with claiming her corresponding funds that are locked in an HTLC on the second blockchain, thus completing the exchange. The inherent transparency of blockchain transactions means that once Bob reveals the preimage in a transaction to claim his funds, this piece of information becomes publicly accessible to Alice, and potentially to other observers on the network, so HTLC also stores the address of Bob and checks if it matches the initiator of withdraw transaction. The public revelation of the preimage ensures that both sides of the atomic swap can be finalized without the need of a centralized coordinator.

Another possible outcome is if a timeout period, enforced by the timelock is exceeded. If Bob, the intended recipient, fails to provide the correct preimage that matches the hashlock within the agreed-upon timelock period, the HTLC is designed to allow Alice, the original sender of the funds, to reclaim the locked cryptocurrency. This timeout mechanism serves as a vital safety feature, preventing the scenario in which funds become locked indefinitely if the transaction cannot be successfully completed for any reason. The specific duration of the timelock is typically negotiated and agreed upon by the parties involved in the transaction. This ensures that the recipient has a reasonable amount of time to fulfill the hashlock condition, while also providing a limit to the period during which the sender's funds are held.[2]

2.2 Atomic escrow

Atomicity, in the context of distributed systems and blockchain, refers to the "all-or-nothing" property of transactions. This property ensures that a series of operations ei-

ther fully completes across all involved parties and systems or entirely reverts, leaving no intermediate or inconsistent state. This characteristic is paramount for maintaining data consistency and preventing partial updates in decentralized environments. For instance, in a cryptocurrency swap, atomicity guarantees that one party does not lose their asset without receiving the counter-asset, thereby eliminating counterparty risk. Academic research[3] models cross-chain swaps as intricate distributed coordination tasks, establishing protocols that guarantee either the complete execution of all swaps if parties adhere to the protocol, or that no conforming party is left in a worse-off position if some parties deviate.

This blockchain-based escrow mechanism offers significant advantages. It dramatically enhances transparency, as all operations and processes are publicly recorded and viewable by participants. It fosters a high level of trust and security by eliminating the need for traditional centralized intermediaries like banks or lawyers, thereby substantially reducing the risk of fraud. Furthermore, by automating the release of funds upon condition fulfillment, it improves efficiency and can help manage transaction costs. However, smart contract escrows are not without drawbacks. They typically incur transaction fees on public blockchains for deployment and execution. Their inherent transparency can lead to privacy concerns, as sensitive information might be exposed. Moreover, the reliance on human programmers introduces a risk of coding errors, which can potentially hamper the contract's execution and negatively impact the transaction.

Chapter 3

Related work

The landscape of cross-chain interoperability is characterized by a variety of solutions, each employing different mechanisms to facilitate asset and data transfer between blockchains. These solutions range from direct asset bridges to more generalized messaging protocols, with ongoing advancements in decentralized escrow implementations that leverage advanced cryptography.

3.1 General applications

3.1.1 Swaps and Liquidity

HTLCs enable seamless asset swaps across different blockchain networks, effectively unifying fragmented liquidity and enhancing trading opportunities on decentralized exchanges. This allows users to trade diverse assets on a single platform, optimizing pricing and reducing slippage, which is a common issue in low-liquidity environments. Cross-chain DEX aggregators, for example, pull liquidity from many networks and optimize trade routing for better outcomes.

3.1.2 Lending and Yield Farming

By allowing assets to move freely across chains, cross-chain escrow enables the utilization of assets from one network (e.g., wrapped Bitcoin on Ethereum) in DeFi protocols on other chains. This significantly expands capital efficiency and opens up new opportunities for users.

3.1.3 NFT Marketplaces and Digital Rights Management

The market for Non-Fungible Tokens and broader digital rights management[4] greatly benefits from cross-chain escrow, which can secure limited-edition NFT sales, provide

immutable proof of authorship and work authenticity, and prevent unauthorized reproduction of digital creative works.

Furthermore, cross-chain escrow facilitates the direct exchange of digital creative works for payment, with assets remaining encrypted and access permissions controlled by the content creator until payment is finalized and verified. This also enables automated royalty payments proportional to secondary sales, creating more equitable compensation models for artists.

3.2 Practical implementations

3.2.1 Bitcoin Lightning Network

The Bitcoin Lightning Network[5] is a second-layer scaling solution built on top of the Bitcoin blockchain, designed to address its inherent limitations in transaction throughput and speed. It operates by creating off-chain payment channels between users, allowing numerous transactions to occur instantly and with minimal fees without requiring each transaction to be recorded on the main Bitcoin ledger. When two parties wish to transact, they open a payment channel by committing a small amount of Bitcoin to a multi-signature address on the blockchain. Within this channel, they can conduct an unlimited number of transactions, updating a shared balance sheet off-chain. Only the opening and final closing of the channel, or in some cases, a dispute resolution, are broadcast and settled on the main Bitcoin blockchain. This mechanism is useful for microtransactions and everyday payments, as it would be impractical and costly to use the main chain for this purpose. By deferring the final settlement to the main chain, the Lightning Network significantly reduces the load on the Bitcoin network, enhancing its scalability and usability for a broader range of applications. A core cryptographic primitive enabling the trustless operation and routing of payments across multiple hops in the Lightning Network is the HTLC.

3.2.2 Smart Contract Escrows

These are escrows that utilize smart contracts that hold funds and release them based on predefined conditions. Kleros Escrow[6], for example, is a decentralized application on Ethereum that supports exchanges of goods, assets, or services. It incorporates a dispute resolution mechanism adjudicated by crowd-sourced jurors. Although primarily operating on Ethereum-based assets, it can facilitate crypto-to-crypto trades, including ETH on Ethereum for SOL on Solana, by leveraging its dispute mechanism. Similarly, canonical cross-chain swaps on Layer 2 solutions like Optimism and Arbitrum utilize native L1-L2 messaging to enhance decentralization and speed up settlements for liquidity providers.

These involve escrow contracts that lock tokens on the Layer 2 chain and release them upon fulfillment of conditions verified on Layer 1.

Chapter 4

Methodology

4.1 Cross-chain HTLC design considerations

4.1.1 Hashlock

Hashing function

As stated in chapter 2.1.2, the crucial part of HTLC, the hashlock, is created using a hash function. When choosing a hash function for cross-chain HTLC escrow, compatibility should be taken into consideration. *Keccak-256* hash function is supported by all three target blockchains[7–9]. Hence, to achieve deterministic results on all blockchains, this function was chosen.

Preimage size

There are different storage constraints on the target blockchains. While it is possible to have a variable size of the preimage, existing limitations, especially on TON, make it an undesirable option, mainly because of the complexities and increased storage usage. The optimal size for the preimage is 32 bytes. It is enough to create a unique combination of bytes, which, combined with a timelock and recipient address constraints built-in to HTLCs, makes it secure enough for escrow purposes. Furthermore, EVM natively supports *bytes32* type[7].

4.1.2 Timelock

Absolute timelock was chosen because it offers superior reliability and determinism in a cross-chain context. This preference emerges from the inherent variability in block production times and block height progression across disparate blockchain architectures. Solana, TON, and EVM chains have distinct consensus mechanisms and block finality characteristics, rendering relative timelocks vulnerable to desynchronization and potential

fund locking or unfair outcomes if one chain's progression significantly deviates from another. By utilizing an absolute timelock, all participating chains reference a universally consistent time, ensuring that the HTLC's expiration and subsequent refund or claim conditions are triggered simultaneously and predictably across all involved ledgers.

4.2 Smart contracts implementations

4.2.1 EVM

Common concepts

In the EVM[10], the state of the blockchain is represented as a collection of accounts. Each account has a unique 20-byte address and acts as an entity on the network. These accounts can send and receive transactions, hold an ETH balance, and, in some cases, store code and data.

There are two main types of accounts on EVM-compatible blockchains: Externally Owned Accounts and Contract Accounts. EOAs are controlled by users via private keys. They can initiate transactions, such as sending ETH or interacting with smart contracts. EOAs do not contain any code. Contract accounts, on the other hand, are controlled by the code they store – their smart contract code. They are created when a smart contract is deployed to the blockchain. Contract accounts cannot initiate transactions on their own, instead, they can only execute operations when they receive a transaction or a call from either an EOA or another contract.

A key characteristic of the EVM account model is that Contract Accounts bundle both code and their own persistent storage (state) within the same account structure. When a transaction interacts with a Contract Account, the EVM executes the contract's code, which can read from, write to, or modify the data stored within that specific contract's storage. Each account, whether EOA or Contract, also maintains a balance in ETH and a nonce, which is a counter used to prevent transaction replay attacks or to track contract creation.

Technology stack

The primary language for writing smart contracts on the Ethereum Virtual Machine is a statically-typed programming language Solidity[11]. Its syntax is significantly influenced by JavaScript, with elements reminiscent of C++ and Python, aiming to provide a familiar programming paradigm for developers. Key features of Solidity include its support for inheritance, libraries, and complex data structures. Concepts such as modifiers, events, and various data locations (e.g., storage, memory, calldata) are integral to Solidity's structure, allowing developers to manage state, log actions, and optimize gas consumption.

The compilation of Solidity code results in EVM bytecode, which is then deployed to the blockchain and executed by nodes participating in the network.

Hardhat[12] is a development framework for compiling, deploying, testing, and debugging Solidity smart contracts. It is built around the concept of "tasks," which are extensible and configurable actions that can be executed from the command line. A core component of Hardhat is its local Ethereum network, which provides a simulated blockchain environment for development purposes. This network allows for rapid iteration during testing and debugging, offering detailed insights into transaction execution, including gas usage, stack traces, and variable inspection. Hardhat's plugin system significantly enhances its capabilities, allowing developers to integrate various tools and functionalities. Notable plugins facilitate tasks such as contract verification on block explorers, gas reporting, and integration with popular testing frameworks like Mocha and Chai. Furthermore, Hardhat has the ability to seamlessly interact with different networks, including public testnets and the Ethereum mainnet. The framework's emphasis on developer experience, through features like console logging within smart contracts and comprehensive error reporting, has made it chosen for this work.

Proposed solution

As seen in Figure 4.1, the EVM HTLC smart contract encapsulates both executable code and storage. Only one contract is required for the system to operate.

Storage:

- **Trades** is a mapping from trade id to the `trade info structure` instance that stores all trades. A mapping represents a hash map data structure in Solidity.

  `Trade info structure` contains details about a particular trade: address of the owner (initiator), address of the recipient, address of the token which is traded, amount of these tokens, timelock, hashlock, completion flag and preimage. When assets are withdrawn, the preimage field of `trade info structure` is set, thus revealing it to the other party of escrow.

- **Current trade id** represents current available id for the trade. Increments with each new created trade.

Smart contract methods:

- **createTrade** is used by the initiator. The initiator sends required amount of assets. Trade info details are provided as arguments to the method. The current trade id variable in storage is incremented should the invoke of this method be successful.

- **withdraw** is used by the recipient. It accepts trade id and preimage. If recipient address and hashlock match provided data, assets locked by this trade are transferred to the recipient.
- **refund** is used by the initiator. Accepts only trade id. The method imposes four requirements to succeed: trade with provided id should exist, the sender of the transaction must be the initiator of the trade, trade should not be completed and timelock should be expired.
- **getPreimage** is a get method that returns preimage associated with a particular trade for the requested trade id if the trade exists and is completed.

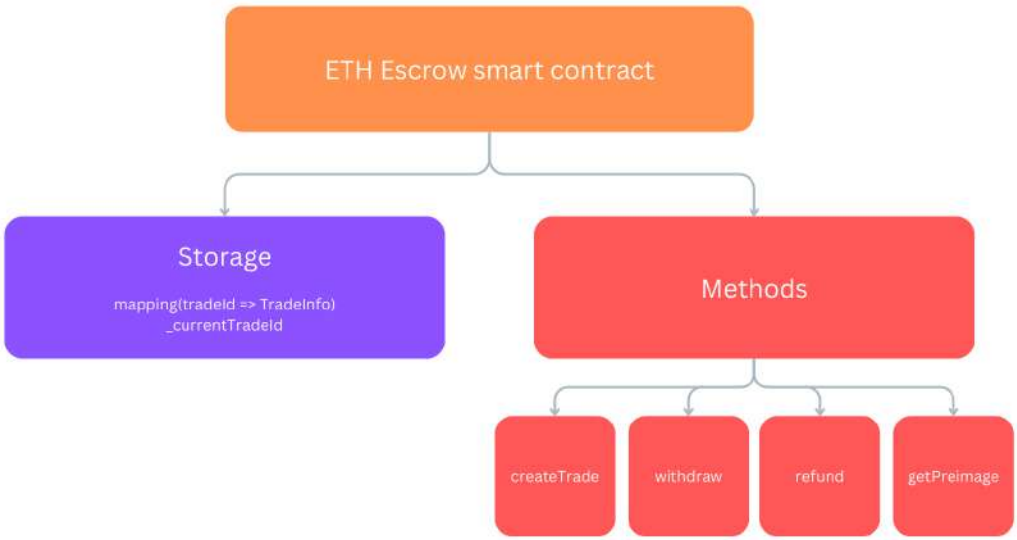

Figure 4.1: EVM smart contract

4.2.2 Solana

Common concepts

The Solana[13] blockchain architecture introduces a distinct account model designed to optimize for high throughput and parallel transaction processing. Unlike traditional blockchain models where contracts often encapsulate both logic and state, Solana employs a stateless program design coupled with a versatile account structure.

At the fundamental level, all interactions and state storage on the Solana network revolve around accounts. An account serves as a data buffer in the network's memory-mapped file system, identified by a unique 256-bit public key (address). Each account possesses metadata, including its current native token balance in lamports, an assigned

owner (another account's address, typically a program), and a flag indicating whether it contains executable code or only data. Crucially, accounts also store an arbitrary data payload, the structure and interpretation of which are dictated by the owning program.

A key architectural decision within Solana is the strict separation between executable accounts (programs) and non-executable accounts (data accounts). Programs are designated as read-only and stateless, they contain the immutable logic but do not hold persistent state themselves. In contrast, data accounts are mutable and serve as the storage for all on-chain state, ranging from user token balances to complex application data structures. This allows the Solana runtime to process transactions involving distinct data accounts in parallel, which is foundational for its high-performance claims. A single program can operate on multiple data accounts within a single transaction, provided that these accounts are explicitly declared beforehand.

The concept of ownership is central to Solana's security and state management. Each account is assigned an owner program upon creation. Only the owner program has the authority to modify the data held within an account it owns. This mechanism ensures that state transitions adhere to the rules defined by the relevant program logic. While anyone can read data from an account and anyone can credit an account with lamports, write permissions are strictly enforced. Furthermore, program derived addresses offer a mechanism for programs to programmatically control accounts without requiring private key signatures, authorizing program to sign any message on the behalf of the account. The bump seed is used in PDA generation and exists to ensure that PDA is valid even if other seeds produce an invalid address. PDA mechanism allows programs to manage assets on the blockchain and enables more sophisticated on-chain protocols and custodial arrangements.

Solana also implements a rent mechanism to ensure the sustainable use of validator resources. Accounts are required to maintain a minimum lamport balance, proportional to the amount of data they store. The rent price is defined by the rent system variable, which is accessible to programs and clients that interact with the blockchain.

Technology stack

Solana's native Rust development paradigm involves writing on-chain programs directly using the Rust programming language, leveraging the solana-program crate. Developers manage data persistence by defining custom data structures that are serialized and deserialized from account data using borsh crate. Programs contain data in Solana Bytecode Format, which is a custom version of eBPF bytecode[14]. While frameworks like Anchor simplify much of this boilerplate, direct native Rust development offers granular control over program logic and optimizations, thus it was chosen to minimize resource consumption. The Solana Program Library further exemplifies this approach, providing a suite of audited, high-performance programs like SPL Token, SPL Associated Token Account

and others that serve as foundational building blocks for the ecosystem.[8]

For testing purposes, the litesvm[15] crate was chosen. Traditional testing methods on Solana often involve solana-program-test or the solana-test-validator, which can be resource-intensive and slower due to their more comprehensive simulation of the full Solana runtime or a full validator node. Litesvm provides a lightweight Solana Virtual Machine simulator, which is used to create an isolated SVM instance within the test environment, allowing for quick compilation and execution of program logic without the overhead of a full blockchain simulation. This significantly reduces test execution time, and makes it a valuable tool for unit and integration testing of Solana applications.

Proposed solution

The Figure 4.2 illustrates the architecture of the solana escrow program. As mentioned in section 4.2.2, Solana programs are stateless, so accounts are used for storage.

Escrow program accounts:

- **Escrow data account** is created for each trade. It is created using PDA and is owned by the escrow program, which is a necessity for assets to be managed by it. The account contains public keys of the initiator, the recipient and token mint as well as amount of tokens, timelock, hashlock, bump seed and creation timestamp. This account is closed upon `withdraw` invoke and the funds held for rent exemption are returned to the initiator.

- **Preimage reveal account** is created if the execution of `withdraw` method succeeds. The purpose of this account is to reveal the preimage. The preimage could be stored in `escrow data account`, but it would be inefficient as storing all trade data for preimage reveal is redundant. The account address is PDA, with an additional seed of `escrow data account` address and creation timestamp which ensures that a new reveal address is generated for each trade and allows two same accounts participating in the trade to perform trades further after the first one. This makes it possible to get the preimage even if account with escrow data is deallocated, which decreases the amount of funds required to be reserved for rent.

Escrow program methods:

- **deposit** creates a trade, requires all data for `escrow data account`. PDA of `escrow data account` should be calculated in advance and passed to the program with other accounts required by the method.

- **withdraw** finalizes the trade, transfers funds to the recipient. It also deallocates `escrow data account` and creates `preimage reveal account` on success. Fails if

given preimage is invalid or if the timeframe of the trade defined by the timelock is expired. Requires an initialized `escrow data account` to be passed alongside other accounts.

- **refund** returns funds to the initiator if the timelock is expired. Additionally, it deallocates the `escrow data account`, thus invalidating the trade.

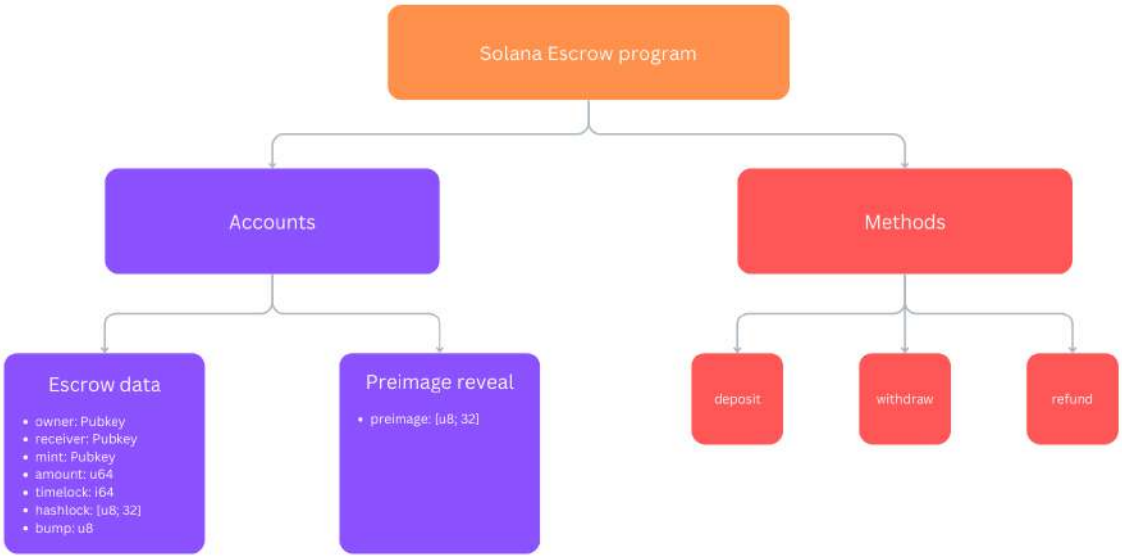

Figure 4.2: Solana program

4.2.3 TON

Common concepts

The Open Network[16] employs a unique concept that differs significantly from many other blockchains, particularly Ethereum. In TON, there is no distinction between an externally owned account and a smart contract, instead, every "account" on TON is essentially a smart contract. This means that even a user's wallet is a smart contract.

Each TON smart contract has fundamental properties: an address, executable code, persistent data, and a balance of Toncoin. Execution of the smart contract can result in the account mutating its storage or sending outgoing messages to other accounts.

A smart contract can be invoked via an external or internal message. The messages are handled by designated functions. The internal message handler is mandatory, but the external one is optional. Internal messages are created only by smart contracts. External messages are sent from outside the blockchain. Therefore, the incoming data in these messages should be thoroughly checked to prevent unexpected behavior.

TON fundamentally adopts the Actor Model, a concurrent computation paradigm, for its smart contracts. This means that instead of direct function calls and shared memory, actors communicate exclusively through asynchronous message passing. When one actor needs to interact with another, it sends a message and the recipient actor processes that message in its own isolated context. This allows TON to split and merge accounts into shards, depending on current network load.

Since each actor maintains its own private state and logic, there is no shared mutable state between actors. This eliminates common concurrency issues such as race conditions and deadlocks, which are prevalent in traditional programming models that rely on locks and shared memory. When an actor receives a message, it processes it sequentially, one at a time, ensuring deterministic behavior within that actor.

4.2.4 Technology stack

As a result of research of the official documentation for The Open Network[9], the following choices were made:

For testing, development and contract deployment Blueprint was chosen. It serves as a comprehensive development environment for TON smart contracts, akin to Hardhat in the EVM ecosystem. It provides an all-in-one suite for various stages of the development lifecycle, including writing, compiling, testing, and deploying smart contracts. Blueprint supports multiple TON smart contract languages, including FunC, Tact, and Tolk, offering flexibility in the choice of programming paradigm. A key feature of Blueprint is its integrated in-process blockchain emulator, the TON Sandbox. This local testing environment allows rapid and isolated execution of smart contracts, simulating on-chain interactions, message passing, and state changes without the overhead of a full network connection. This significantly accelerates the debugging and iteration process. Furthermore, Blueprint facilitates seamless deployment to TON's testnets and mainnet through user-friendly interfaces, often integrating with popular TON wallets for transaction signing and broadcasting using TonConnect. Its project structure and tooling are designed to provide a streamlined workflow, encompassing contract source code management, TypeScript wrapper generation for type-safe interactions, and dedicated directories for tests and deployment scripts, thereby promoting best practices in dApp development on TON.

For smart contract programming language, Tolk was chosen. Tolk is a relatively new, high-level programming language, positioned as a *"next-generation FunC"*. FunC, predecessor of Tolk, is now considered deprecated, which is why Tolk was developed as a replacement. Tolk is retaining FunC's crucial features like inline assembly for low-level interactions with TVM while improving syntax to modern standards. Tolk abstracts away some of the complexities of FunC, introducing features like explicit type declarations and a more intuitive approach to handling cells and slices, which are TON's fundamental data

structures. Storage in TON consists of cells. Each cell can hold up to 1023 bits of data and up to 4 references to other cells[9]. Tolk compiler also embeds the standard library functions, eliminating the need for manual imports of common utilities: an inconvenience that was required in FunC. Tools, such as *@ton/tolk-js*, provide a WASM wrapper for the Tolk compiler, enabling its use in JavaScript/TypeScript environments and are compatible with Blueprint.

Proposed solution

As discussed in section 4.2.3, all entities in TON are smart contracts. When designing TON smart contracts, it is important to keep in mind the sharding features. Therefore, each trade in the escrow is represented with a distinct smart contract.

Storage is represented by a cell, stored data is listed in Figure 4.3. As stated in section 4.2.3, cells can hold up to 1023 bits. Consequently, the preimage is stored in a separate cell referenced by the main storage cell to comply with storage limitations.

Trade creation As each trade is backed by a separate smart contract, there is no method like `deposit` on Solana. Instead, to create a trade one has to deploy escrow smart contract with escrow data defined in storage part of the initial contract state and send required amount of tokens to the contract.

Methods:

- **claim** accepts token wallet address and the preimage. Then performs necessary checks, fails if the preimage is invalid, the sender of the message is not the receiver or if the timelock has expired.
- **refund** accepts only token wallet. Checkes sender address to match owner address and for the timelock to be expired. If successful, closes the contract and returns accumulated contract Toncoin balance to the owner.

Keccak-256 is not included in the standard library, but is supported by TVM. To use it in Tolk, one has to define a function with inline assembly call of `HASHEXT_KECCAK256` TVM opcode[9].

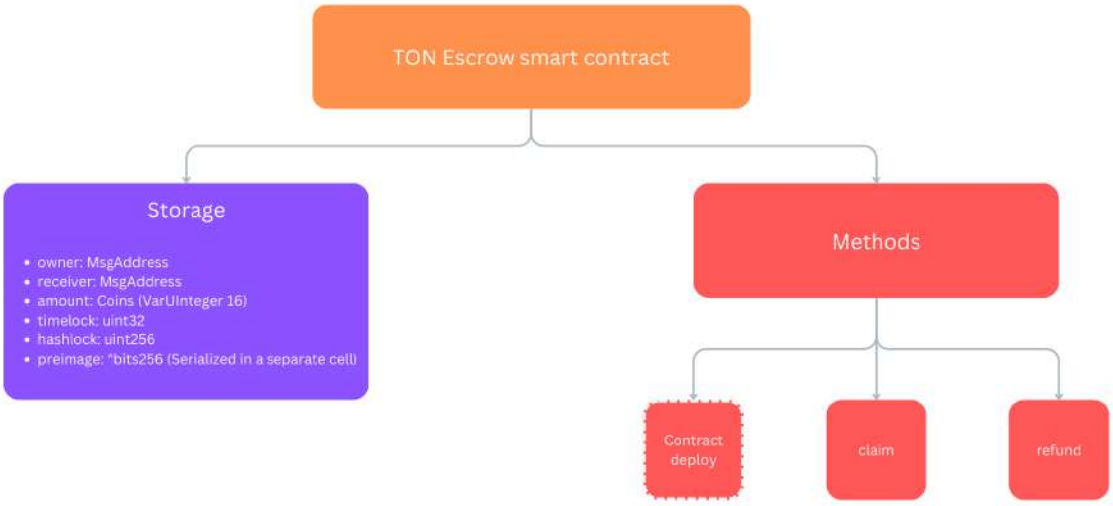

Figure 4.3: TON smart contract

Chapter 5

Evaluation

This chapter presents a comprehensive evaluation of the developed atomic cross-chain escrow solution. The primary goal of this evaluation is to assess the functionality and cost-efficiency of the smart contracts implemented for EVM-compatible blockchains, Solana, and The Open Network. The aim is to demonstrate the viability and effectiveness of the approach in facilitating trustless and atomic cross-chain transactions.

5.1 Evaluation Methodology

The designed evaluation methodology encompasses functional testing and cost assessment across the three supported blockchain platforms.

5.1.1 Objectives

The core objectives of the evaluation are:

Verifying functional correctness is done to confirm that the smart contracts correctly implement the core escrow logic, including asset deposition, locking, release upon confirmation, and refund mechanisms. Crucially, this includes verifying the atomicity of the cross-chain swap, ensuring that transactions either complete fully on all participating chains or fail without loss of funds.

Analyzing economic efficiency determines the transaction costs associated with deploying and interacting with the escrow smart contracts.

5.1.2 Testing Environment

The evaluation was conducted in local and testnet environments to simulate real-world conditions without incurring real financial costs or impacting mainnet operations. The specific environments and tools used were:

EVM Ethereum Sepolia testnet. Smart contracts were deployed and tested using Hardhat framework. Interactions were performed using Ethers.js which is conveniently included in Hardhat.

Solana Solana Devnet. Solana program was deployed and tested using the Solana CLI and litesvm. Interactions were facilitated via the Solana rust client.

TON TON Testnet. Smart contracts were deployed and interacted with using scripts feature of Blueprint.

5.1.3 Evaluation Criteria & Methods

Functional Correctness

A suite of unit and integration tests was developed for each smart contract. These tests covered all defined functions and state transitions. End-to-end tests were performed, simulating various cross-chain swap scenarios.

Scenarios:

- Successful A-to-B swap.
- Successful B-to-A swap.
- Refund A due to timeout (B never deposits).
- Refund B due to timeout (A never withdraws).
- Attempted withdrawal by an unauthorized party.
- Attempted withdrawal after the timelock has expired.
- Verification of state changes and fund movements.

Economic Efficiency

Recorded the transaction fees incurred for each key operation. Direct transaction fees were recorded.

Metrics:

- Deployment cost.
- Deposit cost.
- Withdraw cost.
- Estimated total cost per swap.

5.2 Solution Evaluation

5.2.1 Functional Correctness Assessment

The functional tests yielded positive results across all three platforms.

Core Logic The deposit, withdraw, and refund functions performed as specified in the design. State transitions within the contracts were consistent with the expected escrow workflow. Assets were correctly transferred and secured according to the contract logic.

Atomicity The end-to-end tests successfully demonstrated the atomic nature of the swap. In successful scenarios, both parties received their intended assets. In failure scenarios, all deposited funds were successfully claimed by their original owners.

5.2.2 Economic efficiency

Transaction costs and times varied across platforms, reflecting their underlying architectures. (Note: These are testnet figures and may differ on mainnets under varying load).

EVM (Sepolia)

- Deployment: $\sim$ 892000 gas.
- Create trade: $\sim$ 192000 gas.
- Withdraw: $\sim$ 104000 gas.
- Refund: $\sim$ 81000 gas.
- Transaction times: $\sim$ 15-30 seconds, depending on network congestion and gas price.

Solana (Devnet)

- Deployment: $\sim$ 0.78 SOL.
- Operations: 0.000005 SOL (only one signature required by all methods).
- Transaction times: $\sim$ 1-3 seconds.

TON (Testnet)

- Deployment: $\sim$ 0.05 TON.
- Claim: $\sim$ 0.015 TON.
- Refund: $\sim$ 0.015 TON.

- Transaction times: $\sim$ 10-15 seconds.

The evaluation highlights that while the EVM implementation is functional, Solana and TON offer substantial advantages in terms of both speed and cost for individual transactions. The overall cost-effectiveness largely depends on the contract logic and the prevailing network fees on each platform.

Chapter 6

Conclusions

6.1 Summary of work

The proliferation of diverse blockchain ecosystems has underscored the need for secure and efficient interoperability solutions. Traditional methods often rely on trusted intermediaries or complex bridging mechanisms, introducing points of failure and security vulnerabilities. This work addressed this challenge by focusing on the design, development, and evaluation of an atomic cross-chain escrow system. The primary objective was to create a trustless mechanism enabling users to exchange assets across three distinct blockchain platforms: EVM-compatible chains, Solana, and The Open Network.

To achieve this, three distinct smart contracts were developed, one for each target platform. These contracts implement the core logic for a HTLC-inspired escrow, facilitating the secure deposit, locking, and conditional release or refund of assets. A crucial aspect of design was ensuring atomicity, meaning that cross-chain swaps either execute completely on all involved chains or fail entirely, preventing loss of funds (excluding transaction costs). The implementation considered the unique architectural nuances and programming languages (Solidity, Rust, Tolk).

6.2 Further work

Building upon the foundation laid by this project, several possibilities for future work emerge:

Decentralized Oracles/Relays The escrow relies on off-chain communication between parties of the trade. Integrating decentralized oracle networks or exploring light-client-based verification methods to handle cross-chain communication would significantly enhance decentralization and security.

Expanded Chain Support Extending the system to support additional blockchains would further broaden its applicability and impact.

Formal Verification & Audits Undertaking formal verification where possible and commissioning external security audits are valuable for production-ready product and is crucial for users to trust the service.

Gas/Fee Optimizations Further research into optimizing smart contract design on each platform, especially EVM, could help mitigate costs.

User Interface & SDK Development Creating user-friendly interfaces and Software Development Kits would lower the barrier to entry for users and developers who wish to utilize or integrate the atomic escrow system.

Integration with DeFi Protocols Exploring the integration of this cross-chain escrow with existing Decentralized Finance protocols could unlock novel financial instruments and strategies.

6.3 Concluding remarks

The creation of an atomic cross-chain escrow system supporting EVM, Solana, and TON represents a significant step towards a more interconnected blockchain ecosystem. This paper has successfully demonstrated the technical feasibility of such a system, providing a functional and evaluated solution. Although limitations exist, particularly concerning the cross-chain communication layer, the project lays a strong foundation. The principles and implementations presented herein contribute valuable insights for developers and researchers working towards a truly interoperable, secure, and user-centric decentralized future.

Bibliography

[1] M. Bishnoi, R. Bhatia, A. Jain, and B. Singh. "Hash time locked contract based asset exchange solution for probabilistic public blockchains". In: *Cluster Computing* 25 (June 2022). DOI: 10.1007/s10586-022-03643-x.

[2] Monika et al. "Atomic Cross-Chain Asset Exchange for Ethereum Public Chains". In: *2021 International Conference on Computer Communication and Informatics (ICCCI)*. 2021, pp. 1–4. DOI: 10.1109/ICCCI50826.2021.9402343.

[3] M. Herlihy. "Atomic Cross-Chain Swaps". In: (May 2018). URL: https://arxiv.org/pdf/1801.09515 (visited on 04/05/2025).

[4] R. Ouyang. "HBRO: A Registration Oracle Scheme for Digital Right Management Based on Heterogeneous Blockchains". In: *Communications and Network* 14 (Feb. 2022), pp. 45–67. DOI: 10.4236/cn.2022.141005. URL: https://doi.org/10.4236/cn.2022.141005.

[5] J. Poon and T. Dryja. "The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments". In: (Jan. 2016). URL: https://lightning.network/lightning-network-paper.pdf (visited on 03/21/2025).

[6] C. Lesaege, W. George, and F. Ast. "Kleros". In: (Mar. 2020). URL: https://kleros.io/whitepaper_long_en.pdf (visited on 03/21/2025).

[7] The Solidity Authors. *Solidity documentation v0.8.30*. URL: https://docs.soliditylang.org/en/v0.8.30/ (visited on 04/03/2025).

[8] Solana Foundation. *Solana documentation*. URL: https://solana.com/docs (visited on 04/03/2025).

[9] The Open Network Foundation. *TON Blockchain documentation*. URL: https://docs.ton.org/ (visited on 04/12/2025).

[10] V. Buterin. "A next-generation smart contract and decentralized application platform". In: (Jan. 2014). URL: https://www.weusecoins.com/assets/pdf/library/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf (visited on 04/03/2025).

[11] H. Umucu. “Solidity: Smart Contract Language or Legal Contract Language”. In: (May 2021). URL: https://ssrn.com/abstract=3916072 (visited on 04/03/2025).

[12] Nomic Foundation. *Hardhat documentation*. URL: https://hardhat.org/docs (visited on 04/03/2025).

[13] A. Yakovenko. “Solana: A new architecture for a high performance blockchain v0.8.13”. In: (2017). URL: https://solana.com/solana-whitepaper.pdf (visited on 04/03/2025).

[14] S. Yuan et al. “A Complete Formal Semantics of eBPF Instruction Set Architecture for Solana”. In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (Apr. 2025). DOI: 10.1145/3720414. URL: https://doi.org/10.1145/3720414.

[15] *Litesvm documentation*. Mar. 2024. URL: https://docs.rs/litesvm/latest/litesvm/ (visited on 04/03/2025).

[16] N. Durov. “Telegram Open Network Blockchain”. In: (Feb. 2020). URL: https://docs.ton.org/tblkch.pdf (visited on 04/12/2025).