UNIVERSITY OF TURKU

Formal Verification of Token Standards on Ethereum and Solana: A Comparative Analysis of Fungible Asset Models

Information and communication technologies

Master's thesis

Author:

Mohammad Abubakr

Supervisors:

Sampsa Rauti

Sammani Rajapaksha

17.06.2026

Turku

Master's thesis

Subject: Information and communication technologies
Author: Mohammad Abubakr
Title: Formal Verification of Token Standards on Ethereum and Solana: A Comparative Analysis of Fungible Asset Models
Supervisor(s): Sampsa Rauti, Sammani Rajapaksha
Number of pages: 128
Date: 17.06.2026

Abstract

Blockchain-based token systems are widely used for creation and management of economic assets. This makes their correctness a critical concern for blockchain developers, auditors and security engineers. ERC-20 and SPL tokens are amongst the largest and most used standards that define foundation for token transfer, mint, burning and authorization. However, the architectural and developmental differences in both ecosystems pose distinct challenges in specifying and verifying token correctness. This thesis focuses on invariant-based verification across two heterogeneous blockchain ecosystems.

The thesis develops comparative framework for ERC-20 and SPL token models focusing mainly on correctness properties like balance consistency, supply preservation, transfer validity, minting, and burning constraints. These properties are validated across six tools; Certora, Foundry, KEVM for ERC-20 and Kani, Prusti, and MIRAI for SPL tokens. It follows a qualitative, artefact-driven case study approach that evaluates tools on five benchmarks; verifiability, expressiveness, specification effort, coverage, and soundness of assumptions.

The results depict that the invariant-based verification is a valuable method for checking essential token correctness properties, but its effectiveness depends on underlying architecture, efforts, and testing tool. Ethereum verification is more contract-centric while Solana verification requires additional reasoning about account ownership, signer authority, mint account relationships and runtime assumptions. The tools provide complementary forms of assurance rather than complete standalone guarantees; deductive verification, fuzzing, bytecode-level reasoning, bounded model checking, contract-based verification and static analysis each capture different aspects of correctness

Overall, this thesis shows that formal verification can strengthen confidence in token implementation when the testing procedures are followed strictly with correct specifications and scope. It also highlights the need for layered assurance pipelines that combine formal verification with testing, auditing, and runtime modeling for secure cross-platform token development.

Key words: Formal verification, invariant-based verification, smart contracts, fungible tokens, ERC-20, SPL Token, Ethereum, Solana, blockchain security, token correctness

Term / Abbreviation	Meaning
Blockchain	Distributed ledger for recording verified transactions.
Smart contract	Program code executed on a blockchain.
Token standard	Common rules for token behavior and interfaces.
Fungible token	Token whose units are interchangeable.
ERC-20	Ethereum standard for fungible tokens.
SPL Token	Solana token program for fungible tokens.
Ethereum	Blockchain platform for smart contracts.
Solana	High-performance blockchain with account-based execution.
EVM	Runtime that executes Ethereum smart contract bytecode.
Invariant	Property that must remain true after valid operations.
Formal verification	Mathematical checking of program correctness.
Balance consistency	Correct relationship between balances and supply.
Supply preservation	Transfers do not create or destroy tokens.
Minting	Creating new tokens.
Burning	Destroying existing tokens.
Allowance	Approved amount a spender can transfer.
Delegated transfer	Transfer made by an approved spender or delegate.
Account model	Blockchain state organized around accounts.
Mint account	Solana account storing token supply and mint authority.
Token account	Solana account storing a user's token balance.
Authority	Account allowed to perform privileged actions.
Certora Prover	Solidity verification tool using formal rules.
Foundry	Solidity testing framework with fuzzing support.
KEVM	Formal EVM semantics for bytecode-level reasoning.
Kani	Rust bounded model-checking tool.
Prusti	Rust deductive verification tool.
MIRAI	Rust static analysis tool.
Harness	Wrapper used to test or verify a property.
Bytecode-level verification	Verification of compiled blockchain code.
Source-level verification	Verification of source code.
Runtime abstraction	Simplified model of the execution environment.
Counterexample	Example showing that a property can fail.

Specification	Formal description of expected program behavior.
Precondition	Requirement that must hold before an operation.
Postcondition	Requirement that must hold after an operation.
Assertion	Check that a property is true at a point in code.
Frame preservation	Ensures unrelated state is not changed.
Reentrancy	Repeated external call before state is safely updated.
Access control	Rules limiting who can perform sensitive actions.
PDA	Program-Derived Address used by Solana programs.
CPI	Cross-Program Invocation between Solana programs.
CVL	Certora Verification Language for writing rules.
MIR	Rust Mid-level Intermediate Representation.
BMC	Bounded Model Checking over limited execution paths.
DeFi	Decentralized Finance applications on blockchain.
DApp	Decentralized application using smart contracts.
Gas	Cost unit for Ethereum computation.
Signer	Account that authorizes a Solana transaction.

Table of contents

1 Introduction

1.1 Background and Context

The purpose of blockchain technology was to solve a basic digital problem of ensuring reliable transactions without a centralized authority. In other words, preventing frauds in transactions without a central authority. In essence, Bitcoin demonstrated how cryptographic signatures, public transaction history and proof-of-work achieve this in a peer-to-peer system [1]. This shifted digital trust away from centralized institutions and toward protocol rules, cryptographic verification, and distributed consensus.

However, Blockchain technology has now evolved into more than just cryptocurrency. It has become the basis of decentralized applications, registries of assets, automated agreements and programmable digital systems [2]. Ethereum is playing a very important role in this development, as it presents a general-purpose smart contract platform on which developers can encode application logic on chain [3], [4]. The Solana ecosystem has also emerged as an important smart contract ecosystem with an emphasis on high throughput, low latency, and parallel execution [5]. The architectural differences of Ethereum and Solana mark two influential blockchain platforms.

1.2 Token Standards and Correctness Challenges

Token-based systems rank among the most popular applications of blockchain technology. Fungible tokens support *governance*, *payment, DeFi, digital collectibles and other blockchain-based services* [6]. On Ethereum, ERC-20 is widely used for fungible tokens. Solana offers similar functionality through *token programs* and *account structures* that are built on Solana's account model.

Tokens represent real economic value, so mistakes can have bad consequences. A flaw in the *transfer logic, minting procedure, balance tracking, supply accounting, allowance handling, and authorization rules* can lead to *financial loss* and *faster exploitable behavior*. This makes token correctness both a software engineering concern and a security requirement.

The problem is that correctness problems differ across blockchain ecosystems. Ethereum follows a model for smart contracts that uses a shared state while Solana separates program logic from mutable account data and requires explicit account passing into transactions [4], [5], [8]. Consequently, while similar token goals may be defined in both the Ethereum and

Solana ecosystems, the method of implementation, specification and verification may vary significantly.

1.3 Motivation for Invariant-Based Formal Verification

Blockchain development relies on traditional assurance techniques such as testing, code review, and static analysis. But these techniques cannot guarantee correctness on all reachable states. This limitation is especially important because deployed blockchain code may be difficult to modify once users and assets depend on it.

To overcome this problem, formal verification expresses the intended behavior as exact properties and checks mathematically whether the implementation satisfies them or not [7].

An invariant-based approach is suitable for token systems since tokens are expected to maintain stable properties throughout their lifetime[3]. These *include balance consistency, conservation of supply, valid ownership changes, transfer validity, and proper authorization of privileged actions*.

This thesis focuses on *invariant-based formal verification* of token standards across Ethereum and Solana. The reason behind this is that both ecosystems have similar high-level token behavior , however, the execution and state models of these ecosystems differ, which affects the specification and verification of the token correctness properties. Comparing these ecosystems can clarify how architecture can influence invariant design and formal reasoning.

1.4 Research Aim, Questions, Scope, and Thesis Structure

The thesis's main objective is to investigate how token correctness can be specified and reasoned about across two major blockchain ecosystems with different execution architectures; *Ethereum* and *Solana*. The research mainly focuses on fungible token models and uses invariants as the central link among token standards, implementation behavior and formal verification.

In particular, the thesis has four objectives. First, it determines the key functional correctness properties and security invariants of tokens behavior. Secondly, it looks at how the architectural differences between Ethereum and Solana shape these properties. Third, it investigates how formal verification techniques and tools can be used to reason about these properties. Fourth, it derives practical implications for developers, auditors, and security engineers working with token implementations across heterogeneous blockchain platforms.

The main research questions guiding this thesis are:

1. *RQ1:* What functional correctness properties and security invariants define the behavior of fungible (ERC-20 / SPL) token standards across the Ethereum and Solana ecosystems?
2. *RQ2:* How effectively can selected formal verification tools be used to specify and verify these correctness properties?
3. *RQ3:* What architectural and execution model differences between the EVM and Solana Runtime influence the design, specification, and verification of token invariants?
4. *RQ4:* What are the practical implications, limitations, and trade-offs of applying formal verification across heterogeneous blockchain platforms for developers, auditors, and security engineers?

This thesis covers only *token standards* and *invariant-based correctness arguments*. The goal is not to validate the full consensus protocols of Ethereum or Solana, nor does it aim to conduct a complete security analysis of all decentralized applications. Instead, the focus here is on selected *fungible token models*, their *core correctness properties* and *the verification problems* emerging from their implementation on two different blockchain architectures.
The structure will be as follows

1. *Chapter 2:* Reviews the technical background of Ethereum and Solana, covering token standards, smart contract vulnerabilities, correctness properties, and formal verification tools.
2. *Chapter 3:* Describes the research methodology, detailing case study selection, invariant criteria, verification setup, and evaluation metrics.
3. *Chapter 4:* Defines the fungible token correctness properties and invariants. It establishes the general invariant framework for fungible tokens and then refines it into the concrete invariant sets for the ERC-20 and SPL Token models, including their storage and account layouts.
4. *Chapter 5:* Presents the verification and its results. It reports the Solidity-based verification using Certora, Foundry, and KEVM, the Rust-based verification using Kani, Prusti, and MIRAI, and a consolidated summary of the outcomes across all six tools.
5. *Chapter 6:* Analyzes and discusses the results. It assesses tool effectiveness, compares how the ERC-20 and SPL Token invariants are shaped by the EVM and Solana

runtime, and draws out the practical implications and trade-offs of cross-platform formal verification.

6. *Chapter 7:* Concludes by summarizing findings, outlining research contributions, and suggesting future research directions.

1.5 Declaration on the Use of Generative AI

The work presented in the thesis is the result of experiments and testing the author conducted. Generative AI was used in a limited capacity to help make the experiments better and fast. ChatGPT has been used to create Figures based on my extensive prompts. ChatGPT is also used for refining grammar and polishing the terms.
All submitted parts including the Literature Review, Research methodology, Invariant selection, Result and analysis, Discussion and Conclusion are my own work based on my own research and experiments.

2 Technical Background and Literature Review

This chapter presents the technical background and literature review that is required for the understanding of token correctness and invariant based formal verification across Ethereum and Solana. As this thesis compares the token standards deployed on two different blockchain architectures, the chapter introduces first the general blockchain concepts: *distributed ledger, state transition, consensus, trust minimization and immutability* and then discusses the Ethereum smart contract architecture including the *Ethereum Virtual Machine*, *the global shared state*, *contract storage*, *composability*, and *correctness issues*.

2.1 Fundamentals of Blockchain Systems

Blockchain systems are distributed computing systems that maintain a shared transaction history across multiple participating nodes. In blockchain networks, all transactions are verified and stored on a consistent ledger via *cryptography* and *consensus* rather than *a central authority*. With this arrangement, every participating node despite not fully trusting each other can agree on the order of transactions and their validity.
A blockchain is an append-only ledger where new transactions get grouped, validated and added to the existing history. The integrity of the ledger is maintained by cryptographic links between the blocks and the pre-defined validation rules followed by network participants. The *transparency* and *tamper-resistant properties* of this structure also adds difficulty in *scalability*, *privacy*, *governance* and *software correctness*. According to Conti et al., the secure nature of blockchain comes from the combination of *cryptographic mechanisms* that promote decentralized consensus and the lack of a single trusted party [9].

2.1.1 Distributed Ledgers and State Transition

Transaction history is stored throughout various nodes in a distributed ledger. Each node can independently check whether the transaction follows the rules of the system. When a transaction is accepted, it contributes to a new ledger state. State on an account-based blockchain platform generally refers to account *balances, contract storage, token account states, and other program-controlled data*.
Understanding smart contract correctness relies heavily on state transition concept. A transaction uses the current state as its input data, executes some rules, and generates a new state. The new state will then be included as part of the accepted ledger history only if it is valid. If the transition violates protocol or application rules, it should be rejected or reverted.

For token systems, the state transitions include *balance changes, minting, burning, approval, and delegated transfer*. For example, a transfer operation should decrease the balance of the sender and increase the balance of the receiver by the same amount. The state transition perspective is vital for invariant-based verification. An invariant property is the one which must continue to hold true before and after every valid state transition. For example:

1. Consistent total supply
2. Non-negative balances
3. Valid authorizations

Figure 1 brings these elements together, depicting the layered blockchain architecture in which the distributed ledger, chained blocks, and participating nodes.

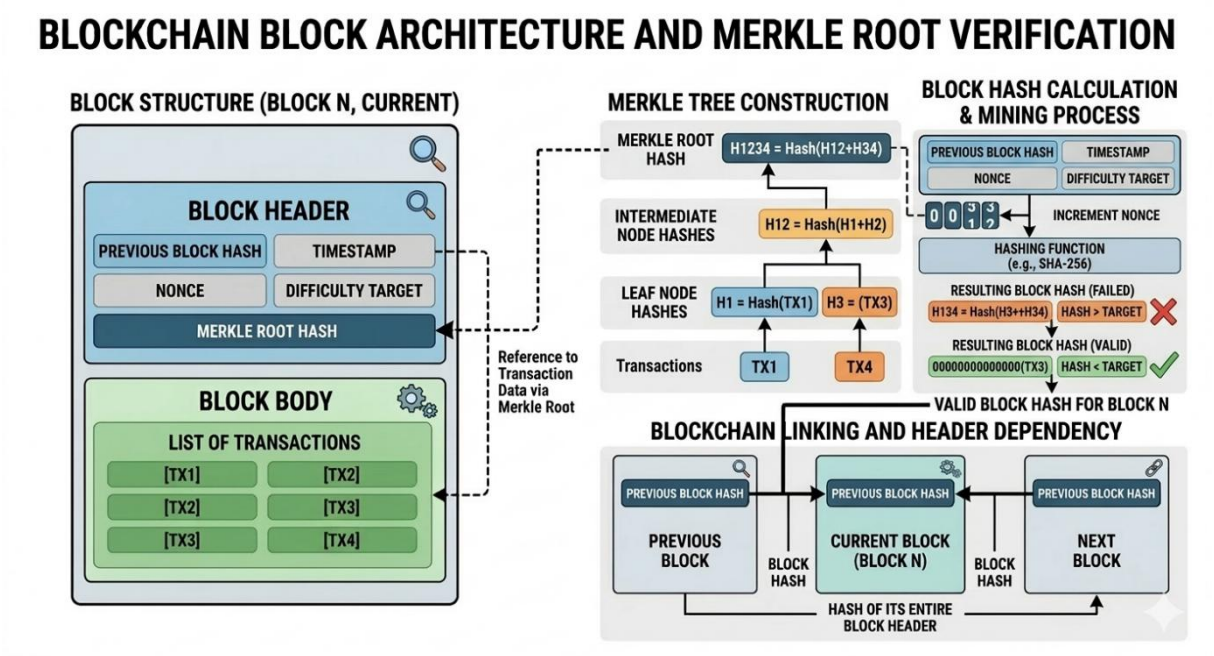

Figure 1: Blockchain Architecture

2.1.2 Consensus, Trust Minimization, and Immutability

The consensus refers to the process through which blockchain participants agree on the transaction order and ledger state. Without agreement, different nodes can have different histories. Therefore, consensus mechanisms allow distributed participants to converge on a shared version of the ledger [1], [2].

Trust Minimization is one of the primary design ambitions of blockchain systems. Users depend on *protocol rules*, *cryptographic verification*, and *execution* that is publicly verifiable rather than a centralized institution. Wüst and Gervais argue that blockchain systems are most

appropriate when multiple parties need to share state but do not want to rely on a trusted central authority [10]. The observation has relevance for token systems as rules for the ownership and transfer of tokens are enforced by shared infrastructure, not a single database administrator.

Another critical property of blockchain systems is Immutability. Once the transaction is executed and widely accepted by the network, it becomes immutable [1]. Although it improves *reliability* and *user trust*, it also amplifies the consequences of software faults. If a smart contract or token program is deployed incorrectly, the flaw may remain exploitable unless the system is able to upgrade or migrate.

The correctness of blockchain is therefore a bigger issue than typical application correctness. In many conventional software setups, debugging can be done by updating the server-side application code. However, changing deployed code and recorded state is difficult in blockchain systems [15]. Thus, *pre-deployment assurance*, *testing*, *auditing*, and *formal verification* are essential [7].

2.1.3 Benefits and Limitations of Blockchain Systems

This section identifies the main strengths and weaknesses of blockchain systems from the perspective of software correctness and verification. Table 1 summarizes these benefits and limitations to show why formal assurance is important for blockchain-based token systems.

Aspect	Benefit	Limitation
Decentralization	Reduces reliance on central authorities [10]	Makes governance and upgrades more complex
Transparency	Allows public verification of transactions	May expose transaction patterns [9]
Immutability	Improves auditability and tamper resistance [1]	Bugs may be difficult to fix after deployment [5]
Programmability	Enables smart contracts and token systems	Smart contract bugs can cause financial loss [7]
Composability	Allows applications to interact with each other	Creates complex interaction and attack surfaces [15]
Consensus	Provides agreement on transaction order	Can introduce scalability and latency trade-offs [2]

Table 1: Benefits and limitations of blockchain systems

2.2 Ethereum Smart Contract Model

Ethereum is a programmable blockchain that supports smart contracts for decentralized applications (DApps). A smart contract is executable code deployed on-chain and associated with an Ethereum address. Once deployed, users and other contracts can interact with it by sending transactions or message calls. According to the Ethereum documentation, a smart contract is a kind of Ethereum account that can hold a balance and execute code when triggered by transactions [11].

The Ethereum smart contract model is of particular importance for this thesis due to the fact that ERC-20 tokens are implemented as smart contracts. The correctness of token contracts' execution relies on the functioning of underlying *Ethereum Virtual Machine, the storage of tokens state* in persistent storage, and *contract-to-contract messaging* exchanges.

2.2.1 Ethereum Virtual Machine and Smart Contract Execution

The Ethereum Virtual Machine executes Smart Contracts and works on the Ethereum Blockchain. Solidity is a language to create smart contracts for diverse Ethereum applications [11]. When a transaction calls a contract function, Ethereum nodes run the corresponding *bytecode* and update the global state if the execution is successful [24].

The execution provided by the EVM is deterministic therefore, every honest node executing *same transaction* from the *same initial state* should return the *same result [4]*. This determinism is needed so that every node comes to agreement on the resulting state [24].

Execution is also constrained by gas, which limits computation and prevents unbounded execution [4]. Every operation on the EVM has a gas cost; hence, the transaction must provide enough gas for successful execution.

For token contracts, EVM execution determines how functions such as *transfer*, *approve*, *transferFrom*, *mint*, and *burn* modify contract storage. Correct execution requires that state updates occur only after required checks are satisfied [15]. For example, an ERC-20 transfer should verify that the sender has sufficient balance before updating balances [13]. Figure 2 illustrates this execution path, showing how a transaction invokes a contract function, the EVM executes the corresponding bytecode, and the global state is updated only when execution succeeds.

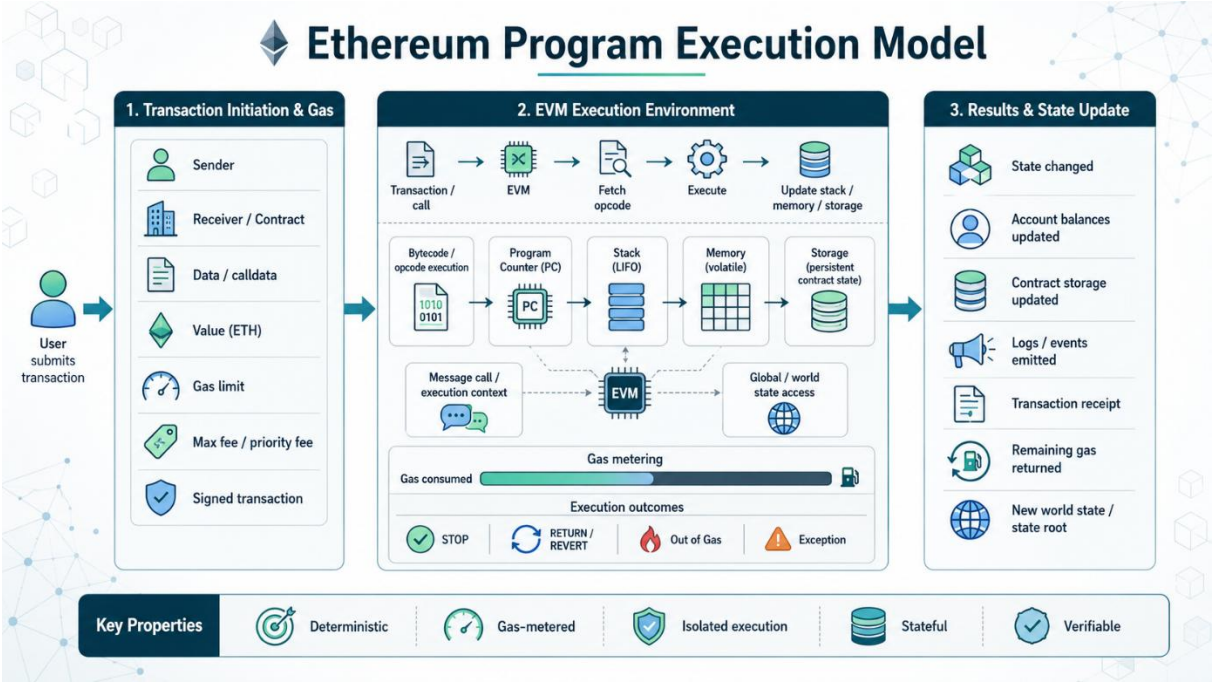

Figure 2: Simplified Ethereum program execution model.

2.2.2 Global Shared State and Contract Storage

Ethereum keeps an account-based global shared state. This includes externally owned accounts controlled by a private key and contract accounts controlled by the deployed code. Each account has fields for *balance*, *nonce*, *code*, and *storage* associated with it. Ethereum's documentation explains that Ethereum state is made up of accounts, and state transitions occur through transfers of value and information between accounts [12].

The stored data of a contract is permanent and is used to track state that is specific to applications. In token contracts, this storage contains the core data structures that define token correctness. For tokens of the ERC-20, balance is done by mapping from addresses to token amounts, whereas allowance is done through a nested mapping from owner to approved spender.

The invariant based verification gets directly impacted by this storage model. Due to the dependence of token correctness on multiple storage variables, it may happen that a function is incorrect even though each update seems plausibly correct. A violation of supply conservation would occur if an ERC-20 transfer updates the recipient balance but fails to decrease the sender balance.

2.2.3 Composability and Contract Interaction

Ethereum smart contracts are like building blocks that can fit together with other contracts, forming larger decentralized applications. One of the reasons token standards became important is composability [14]. This means *wallets*, *swaps*, *exchanges*, *auctions*, *lending protocols* and other *applications* can interact with those tokens in a standard and a consistent way [13].

The ERC-20 standard defines a common interface for fungible tokens, including functions such as *totalSupply*, *balanceOf*, *transfer*, *allowance*, *approve*, and *transferFrom* [13]. The standard allows delegated spending, where the owner of a token approves a different address to spend their tokens. According to EIP-20 specification, the standard allows tokens to be transferred and approved for another on-chain third party [13].

Composability enhances the utility of token standards, but it also enhances correctness risks [15]. Multiple external systems can rely on a token to work in a specific way. An improper implementation of the *transfer*, *approval* or *allowance rules* in a token contract can lead to not just an error in the token but also can impact each and every application utilizing that token. Token verification must evaluate how a function works and also whether the function satisfies the global invariant across contract calls [7]. The most important ones are ERC-20 transfers must not change total supply; delegated transfers must not exceed allowances [31].

2.2.4 Ethereum-Specific Correctness Concerns

Correctness issues particular to Ethereum arise from factors such as the EVM execution model, Solidity programming patterns, gas constraints, and contract-to-contract interaction [15]. Common vulnerability patterns include *re-entrancy, missing access control, arithmetic* mistakes*, unchecked external calls*, *incorrect authorization*, and *inconsistent state updates* [31]. Early work by Luu et al. showed that Ethereum smart contracts can suffer from systematic execution-level security problems, demonstrating the need for automated and formal analysis of contract behavior [16]. Atzei et al. provide a taxonomy of Ethereum smart contract vulnerabilities and show how programming mistakes can lead to exploitable behavior [15].

Vulnerabilities of token systems often show up as correctness property violations. An incorrect balance update in an ERC-20 contract is a violation of supply conservation [15]. Poor access control could potentially permit unauthorized minting and burning [30]. The aforementioned issues are closely related to invariants. A token contract is not correct simply

because the individual functions do not crash but it must also preserve the correct relationships between balances, supply, ownership, approvals, and authorization rules [7]. As a result, formal verification is useful since it can write these expected properties as specifications and check them against the possible behavior of contracts [54].

2.3 Solana Program and Runtime Model

Solana is a high-performance blockchain platform that is designed around parallel transaction execution, and Solana has an account-based runtime model [5]. Solana distinguishes executable program logic from the accounts that contain mutable data. This is different from Ethereum, which has both code and permanent storage. This distinction is important for token verification because correctness depends not only on the program logic but also on whether the correct accounts are passed, validated, owned, and updated during execution [32].
All on-chain state is saved in accounts in Solana. The accounts are the basic unit of data for storing state and is addressed by a public key and includes *lamports*, *owner*, *executable*, and *arbitrary data* [17]. Each transaction, along with one or more instructions, specifies the program to run, which account to call and data to send [18]. If any instruction fails during the transaction execution, the entire transaction will fail, and state changes will be rolled back [18].

2.3.1 Solana Runtime and Account-Based Architecture

The runtime model of Solana is based on explicit access to accounts [17]. A transaction must specify the accounts it will read or write before executing. As a result, the runtime can determine which transactions may execute simultaneously. Transactions that only access read-only accounts can be run concurrently while transactions that require writeable access to the same account must be serialized [19].
The execution of a global shared state differs from Ethereum. An Ethereum contract can read and write to its own storage as well as call other contracts [4]. In Solana, the transaction must include accounts that the program run on, so the correctness relies largely on account validation. A program must verify that the accounts provided are of the *expected type*, owned by the *expected program*, *initialized correctly*, and *signed by the expected signer* [22]. Figure 3 depicts this model, showing how a transaction supplies the accounts a program operates on and how the program validates those accounts before it reads or writes their data.

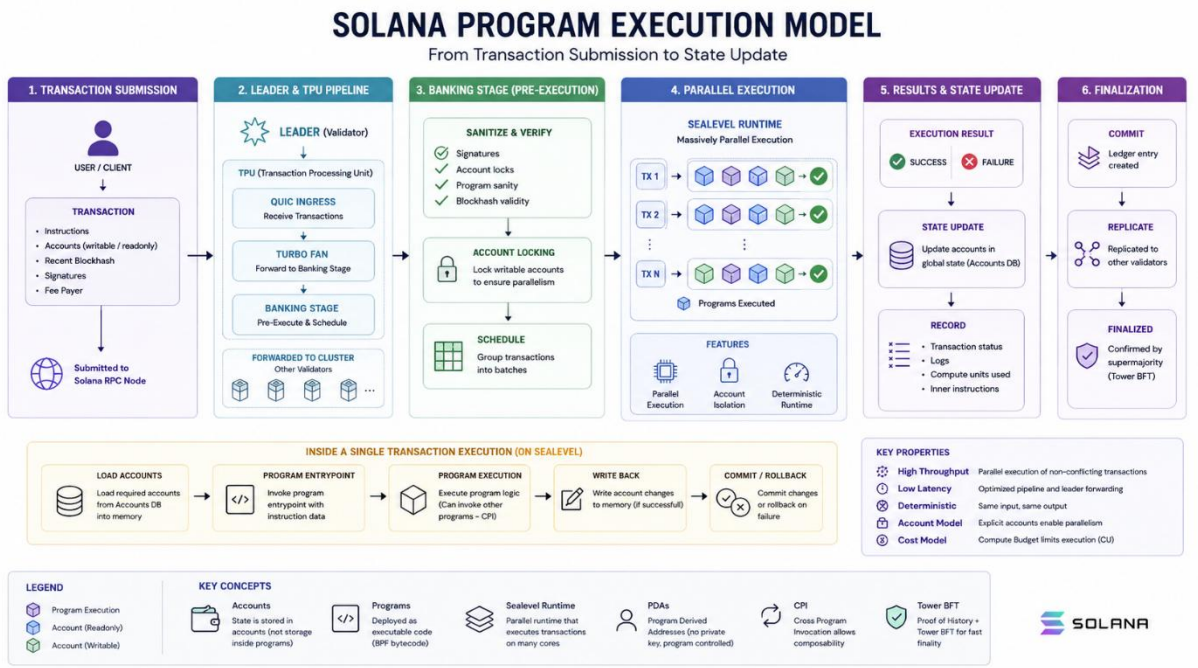

Figure 3: Simplified Solana program execution model.

2.3.2 Program Logic and Mutable Account Data

Solana programs are stateless because the executable code is stored apart from the accounts containing application data [17]. A program's ability to read and alter account data relies on if the account into the instruction and has the authority to carry out the modification. Therefore, account ownership and account mutability are central correctness concerns in the Solana account model [32]. For token systems, this means that *balances, mint information, token accounts, owners, and authorities* are represented through separate accounts rather than through mappings inside a single smart contract. Standard SPL tokens use mints and token accounts to represent fungible token supply and balances [21]. Invariant specification is affected by this design. An example of this could be an ERC-20 balance invariant being written over a mapping inside some contract while an SPL token balance invariant must be about token accounts associated with the mint.

2.3.3 Parallel Execution and Account Access Control

One of Solana's key architectural designs is its *parallel execution model* [5]. When transactions declare their account read and write sets in advance, the runtime can run concurrent transactions without conflicts. This improves throughput, but it also places more responsibility on program and transaction design. The essential point for correctness is that

the access to the account must be explicit and checked. If a program assumes a supplied account is valid without checking ownership, signer status, or expected structure, an attacker may be able to pass a malicious or incorrect account. Solana security guidance identifies that *missing ownership checks, missing signer checks, incorrect PDA validation*, and *improper account validation* are all important classes of vulnerabilities [22].

2.3.4 Solana-Specific Correctness Concerns

The concerns regarding correctness specific to Solana are highly connected with the account model of Solana [8], [32]. Common issues include

1. Missing signer checks
2. Missing ownership checks
3. Incorrect account deserialization
4. Unchecked account relationships
5. Invalid program-derived address validation
6. Confusion between token accounts, mint accounts, and authority accounts

These problems can break token invariants even when the main program logic appears correct [8]. As an example, a program that fails to verify that the token account belongs to the expected mint might end up accepting a bad token account [21]. If it does not check the right authority, unauthorized transfers or minting may be possible [22].

2.4 Comparative View of Ethereum and Solana

Ethereum and Solana both provide functionality for programmable blockchain applications [5]. However, their execution and state models differ significantly. Such differences change the way one implements the token standard, and expresses correctness properties, which will affect their formal verification [7].

The Ethereum platform utilizes a contract-based model which allows smart contracts to deploy code and state. The token state is often stored through mappings and variables within the token contract itself [13]. Solana employs an account-centerd structure where executable programs work on externally supplied accounts. The distribution of token state occurs in mint accounts, token accounts, and authority fields [21].

2.4.1 Execution Environment Differences

Ethereum executes transaction processing of a smart contract bytecode sequentially within the Ethereum Virtual Machine [4]. A contract function is invoked by a transaction and if the execution is successful then the internal storage of the contract is updated. This model makes contract-level reasoning relatively direct because the relevant state is often located inside the contract [24].

Solana's runtime executes programs over accounts explicitly supplied [5]. Transactions allow pre-specification of account access so that they can be executed in parallel when the read/write sets at the accounts do not conflict [19]. Solana verification must not only reason about program instructions, but also the correctness of the accounts supplied to these instructions [32].

The main verification implication is that Ethereum invariants often focus on contract storage, while Solana invariants must also include account validity, account ownership, signer authority, and relationships among multiple accounts [32].

2.4.2 State Representation Differences

Ethereum token state is usually represented through mappings inside smart contract storage. For instance, contract mappings store ERC-20 balances and allowances that can be accessed on demand. The state of Solana token is spread among various accounts. A fungible token system may comprise a mint account, multiple token accounts, and authorities.

This representation gap shapes how *invariants* are expressed. A *balance invariant* for *Ethereum* is defined over the entries of a single contract mapping [13], whereas a comparable *invariant* for *Solana* is defined over a set of token accounts tied together by their mint [21]. The *Solana* definition therefore carries an additional requirement: every token account must belong to the expected mint before its *balance* can be counted [32]. Thus, the same correctness concept takes a different specification form on each platform.

2.4.3 Implications for Token Verification

The architecture of Ethereum and Solana makes token verification different. Validation in Ethereum often revolves around validating contract functions, updating stored values, and sequences of the calls [7]. The verification of Solana must also consider account passing, account ownership signer limits, and program derived address [32]. The invariants of an ecosystem for fungible tokens include balance consistency, supply conservation and

authorization correctness [33]. However, the way these invariants are expressed differs. In general, ERC-20 invariants are written over contract storage, while SPL token invariants must also consider mint accounts and token accounts [21]. Table 2 summarizes the main execution, state, token-state, and verification differences between Ethereum and Solana.

Dimension	Ethereum	Solana
Execution model	EVM executes smart contract bytecode	Runtime executes programs over declared accounts
State model	Contract-centered global shared state	Account-centered distributed state
Token state	Usually stored in contract mappings	Stored across mint, token, authority accounts
Transaction execution	Mostly sequential within EVM execution	Parallel execution possible for non-conflicting accounts
Verification focus	Contract storage, function behavior, call sequences	Account validity, ownership, signer checks, account relationships
Common correctness risks	Reentrancy, allowance bugs, access control errors	Missing signer checks, missing ownership checks, invalid account relationships

Table 2: Comparison of Ethereum and Solana execution and state models

2.5 Token Standards in Blockchain Ecosystems

Token standards provide common interfaces and behavior expectations for blockchain-based assets [14]. In this thesis, they matter because they define what operations' correctness must be checked later by invariants. The ERC-20 and SPL Token are two fungible token models for Ethereum and Solana blockchain models respectively.

2.5.1 Fungible Token Standards: ERC-20 and SPL Token

ERC-20 is a standard interface adopted by Ethereum for fungible tokens. This includes methods to verify the total supply, check balances, transfer tokens, approve spending, and transfer tokens with approval [13]. From a correctness viewpoint the most important properties are balance preservation, valid allowance handling [31], and controlled supply changes [30].

The SPL Token model offers fungible-token features on Solana, similar to ERC20. However, unlike the contract mappings used by ERC20, its state uses mint accounts and token accounts. Therefore, correctness of SPL depends on the balance and supply rules, as well as the validity of account relationships [33], like whether a token account belongs to an expected mint [32]. Although they have different structural characteristics, both standards include the same fundamental operations, namely minting, burning, transferring, and delegating spending, and it is these operations that determine the surface area an *invariant* must constrain [13]. In each case, the common purpose is that balances, total supply, and authorization remain consistent after all possible valid operations [33]. The only difference is the state onto which these properties are mapped: contract mappings on *Ethereum* and account structures on *Solana* [21]. For this reason, a paired-invariant framework is viable.

2.5.2 Token Interoperability and Ecosystem Adoption

Token standards allow interoperability [13]. Token implementations expose predictable interfaces which allow wallets, exchanges, marketplaces, bridges and Dapps to support many assets [6]. This interoperability is useful but can magnify the effects of implementation errors. If a token behaves in a non-standard way, external applications could incorrectly handle balances, transfers, or ownership [14]. Consequently, we should view token standards not just as definitions of interfaces, but also as informal correctness contracts. They specify what other applications expect from a token implementation.[14]

2.6 Smart Contract Vulnerabilities and Token Failures

2.6.1 Ethereum Vulnerability Patterns

Ethereum token contracts may suffer from several vulnerability patterns. Atzei et al. classify important Ethereum smart contract attacks and show how implementation mistakes can lead to exploitable contract behavior [15]. In token contracts, the most relevant patterns include.

1. A contract can invoke an external contract before finalizing its own state changes. Should the outer contract call back into the inner contract, it is possible to affect *token balances, allowances, or supply records* before the first execution finishes [16].
2. Privilege functions like minting, burning, pausing, or transferring administrative ownership may not be properly restricted. Unauthorized users have the ability to alter token supply or contract configuration [30].

3. A token contract may not check to see if the caller is the owner, approved spender, approved operator, or authorized minter. Unauthorized transfers of ERC-20 can take place.
4. Implementation of ERC-20 delegated transfer logic may be wrong. For instance, transferFrom may reduce allowance fail or may allow spending more than the approved amount.[31]

2.6.2 Solana Vulnerability Patterns

Solana vulnerabilities are closely related to account validation [32]. Because Solana programs run on accounts provided by transactions, they must verify that every account has the expected owner, signer status, type, mutability, and relationship to other accounts. According to Smolka et al. Solana's programming model comes with a set of new vulnerability patterns, unlike Ethereum's contract-storage model [8]. According to Anchor's security guidance, the absence of checks on signers and ownership ranks among common risks affecting Solana programs [22].

1. A program might not check for the required signature of authority of the transaction. Unauthorized transfers, minting, or burning may be allowed [64].
2. A program may not confirm that an account belongs to the expected program. An attacker may bypass a phony or rogue account through this [34].
3. A token account, mint account, or authority account may not be checked against the expected asset. This can break SPL token consistency.
4. It is possible that the appropriate addresses and checks are not derived from a program. This can let attackers replace authorized accounts with unaccounted-for program-controlled accounts.
5. Without checking that it is the correct writable account, a program may write to an account. This can lead to invalid updates or unintended states.

2.6.3 Relationship Between Vulnerabilities and Broken Invariants

A vulnerability becomes relevant to formal verification when it allows the token system to reach an invalid state. This invalid state can be expressed as an invariant violation. For example.

1. Unauthorized minting violates supply constraints [30]
2. Incorrect transfer logic violates balance consistency
3. Allowance misuse violates delegated authorization rules [31]

ERC-20 inconsistency can also be understood as an invariant-level failure when the observable transfer behavior, emitted events, and actual balance changes do not agree [14]. This connection provides the bridge between vulnerability analysis and the invariant framework in Chapter 4. The goal of formal verification is to not only detect known vulnerabilities, but also to prove that important correctness properties hold over all valid executions considered by the verification model [7]. Table 3 links common token failure scenarios to their likely causes and corresponding verification targets.

Failure Scenario	Possible Cause	Verification Target
Unauthorized minting	Missing access-control or authority check	Only authorized minter can increase supply
Incorrect transfer update	One-sided or inconsistent balance update	Sender and receiver balances change consistently
Allowance misuse	Delegated transfer does not reduce or respect allowance	Spender cannot exceed approved allowance
Approval persists incorrectly	Approval not cleared after ownership change	Approval state remains valid only for current owner
Invalid Solana account use	Missing signer, owner, or PDA validation	Program accepts only valid authorized accounts

Table 3: Token failure scenarios and corresponding verification targets

2.7 Formal Verification of Smart Contracts

Formal verification is a verification method that mathematically verifies whether a program conforms to its specifications or not [47]. It adopts a procedure-oriented approach in a formal way. Formal verification of smart contracts is important since deployed code on the blockchain may manage financial assets and may be impossible to amend post-deployment [15]. Conventional verification techniques can demonstrate the presence of bugs for some inputs but cannot prove that essential properties hold for all possible executions [16]. Bhargavan et al. describe formal verification of smart contracts as a way to reason about both runtime safety and functional correctness of contract behavior [7]. This notion bears concrete relevance to token standards since we can express many token requirements as correctness

properties. For instance, in a fungible token, the relationship of balances and supply must be maintained under all valid state transitions and actions [33].

2.7.1 Overview of Formal Methods

Formal methods refer to mathematically based techniques to specify, analyze, and verify the system or software behavior [47]. The most relevant formal methods in smart contracts verification are model checking, deductive verification, symbolic execution, abstract interpretation, and theorem proving [16]. The aim of this thesis is not to prove the blockchain protocol correct. Rather, it focuses on the checking of chosen correctness properties of token implementations. These properties are written as invariants, preconditions, postconditions, assertions or rules, depending on the tool that is used for investigation [7].
In token systems, we can do with formal verification because the same actions might be executed many times from various states and users. A transfer, approval, mint, or burn operation should preserve correctness regardless of the specific account values, provided that the required assumptions are satisfied [7].

2.7.2 Formal Specification of Smart Contract Behavior

A formal specification describes what the program should do [54]. Specifications for smart contracts can include constraints and relationships among the storage variables or accounts, safety properties, and authorization rules [30].
For token standards, common specification examples include:

1. A transfer must not create or destroy tokens
2. A sender cannot transfer more tokens than they own
3. Only authorized accounts can mint new tokens
4. An approved spender cannot exceed the approved allowance

Essentially, without a description of what the *right* behavior ought to be, the verification tools cannot check for correctness automatically. The specification acts as the formal version of the token standard's correctness expectations.

2.7.3 Source-Level and Bytecode-Level Verification

Smart contract verification can be done at different abstraction levels. Source-level verification validates characteristics against the source code reflecting program behavior. This

approach is usually easier for developers because the specification is closer to the programming language and application logic [25].

Bytecode-level verification cross-examines the program that executes on chain [24]. It gives a stronger confidence that deployed code satisfies needed properties, since it does not rely solely on the source level view of the program. Nevertheless, the verification at the bytecode level is generally more difficult as the reasoning occurs at a level closer to the execution semantics of the virtual machine [40].

KEVM is a form of reasoning with Ethereum bytecode. According to Hildenbrandt et al., KEVM is an executable formal specification of the EVM bytecode language by the K framework, which they utilize to verify ERC-20 implementations [24]. This makes KEVM especially relevant for this thesis because it connects formal EVM semantics with token-standard verification.

In the case of Rust smart contracts or programs, verification is more likely to be at-source or intermediate-representation level. *Kani, Prusti, and MIRAI* are tools that reason about Rust code using model checking, deductive verification, and abstract interpretation [28], [29].

2.7.4 Strengths and Limitations of Formal Verification

Compared to ordinary testing, formal verification gives better assurances because it actually proves that certain property holds under the assumptions and boundaries of the verification model. This is useful for token contracts because critical properties such as *supply preservation and allowance correctness* [33] must hold across many possible executions, not only in manually selected test cases [16]. Nonetheless, the formal verification has limitations. The verification result is only as strong as the specification. If a specification misses a key property, the tool can prove the implementation correct with regard to that incomplete specification. Additionally, verifying can take a lot of effort, which is particularly true when modeling external calls, account relationships, arithmetic constraints or when dealing with more complex contracts interactions [32]. Also, some of the tools may not be able to scale well for large systems or may have simplified versions of the actual deployment environment [47]. Thus, we should not consider formal verification as a replacement of testing, auditing, code review and so on but rather as their complement. The thesis taps into formal verification to reason about invariant preservation in selected token behaviors. Table 4 summarizes the formal verification concepts used later to define token correctness properties.

Concept	Meaning in Verification	Token-Related Example
Invariant	Property that should always hold	Total supply remains consistent with balances
Precondition	Requirement before an operation executes	Sender must have sufficient balance
Postcondition	Required results after execution	Recipient balance increases by transferred amount
Assertion	Local property checked at a program point	Allowance never becomes negative
Authorization rule	Restriction on who may perform an action	Only mint authority can mint tokens

Table 4: Formal verification concepts and their role in token correctness

Existing Verification Tools for Blockchain Systems

This section briefly reviews selected tools relevant to the thesis. The aim is not to provide a complete survey of all verification tools, but to identify tools that can support invariant-based reasoning for Solidity and Rust-based token implementations.

2.7.5 Solidity Verification Tools: Certora, Foundry, and KEVM

For Solidity contracts, three relevant tools are Certora, Foundry, and KEVM.

1. Certora Prover verifies smart contracts against rules written in the Certora Verification Language. The tool can be utilized to state contract-level properties such as access-control rules, balance restrictions and token invariants [25]. The Certora documentation states that Prover checks if a smart contract satisfies CVL written rules.
2. Foundry is primarily a Solidity development and testing framework but also supports fuzzing and invariant testing [26].
3. KEVM offers a formally executable semantics of the EVM which enables reasoning at the bytecode level. It is relevant when verification should be tied closely to EVM execution semantics rather than solely to Solidity source code [24].

2.7.6 Rust-Based Verification Tools: Kani, Prusti, and MIRAI

For Rust-based programs, three relevant tools are Kani, Prusti, and MIRAI.

1. Kani is a precise bit-level model checker for Rust, and it uses proof harnesses that analyze properties of Rust programs, and it can either prove a property, find a counterexample, or run out of resources [27]. Because of this, it works well for smaller Rust functions or isolated token logic.
2. Prusti is a deductive verifier for Rust based on the Viper verification infrastructure [44]. Astrauskas et al. describe Prusti as a formal verification tool that can verify rich correctness properties of Rust programs with relatively modest annotation overhead [28].
3. MIRAI is an abstract interpreter for Rust's mid-level intermediate representation. It is helpful in identifying certain classes of Rust bugs and reasoning about potential runtime failures [29]. It’s repository states that MIRAI is an abstract interpret for Rust MIR.

The selected tools differ in verification level, specification style, and practical usability. Certora is useful for writing high-level contract rules, Foundry is practical for invariant-style testing, and KEVM provides a formal EVM semantics. Kani, Prusti, and MIRAI are relevant for Rust-based verification, but their direct application to Solana programs may require simplified models of account structures and runtime behavior.

2.8 Summary of Research Gaps

The literature reviewed in this chapter shows that blockchain, token standards, smart contracts vulnerability, and formal verification have been studied with from different perspectives. However, several gaps remain when these areas are considered together in the context of token correctness across Ethereum and Solana.

To start with, the existing work often studies Ethereum and Solana in isolation. There is an extensive literature on smart contract vulnerabilities, EVM execution, ERC-20 tokens, and formal verification tools [15]. Solana research, being a newer and less popular blockchain ecosystem, primarily focuses on runtime architecture, fuzzing, and account-based vulnerabilities [8], [32]. Not much research has been done yet on how Ethereum’s design and Solana’s design affect the verification of these similar properties. Secondly, token standards are often discussed as either an interface specification or an ecosystem component but not for their invariant structure [13]. While ERC-20 and SPL Token are intended to provide correct behavior of assets, the manner in which the correctness properties are represented varies across platforms.

Third, many vulnerability studies describe attacks and implementation mistakes [15], but they do not always connect these vulnerabilities to formal token invariants [60]. For this thesis, the important point is not only that a vulnerability exists, but also which correctness property it violates. For example, *unauthorized minting* breaks supply constraints and *allowance misuse* breaks delegated authorization [31].

Fourth, there is a huge disparity in the language used in the verification tools, expression styles and abstraction levels when implemented. Solidity tools like Certora, Foundry and KEVM provide different kinds of reasoning contract-level, invariant based, and bytecode level reasoning. Tools that have a rust orientation like Kani, Prusti, and MIRAI provide verification support for Rust code. But direct application on Solana programs may need additional modeling of accounts, authorities, runtime constraints, etc [32]. There is a real gap in understanding how verification effort and tool suitability differ between the two ecosystems of Ethereum and Solana [57].

The issues listed above highlight another way in which the challenge of verifying fungible tokens based on invariants goes beyond those already mentioned. Invariant properties for tokens generally hold across many different functions rather than for only a single function, and they depend on multiple stored values and addresses to remain valid; thus, it is entirely possible for a function to appear correct when examined individually yet still break a global invariant property such as total supply conservation [7]. The correctness of an invariant must hold for every state that a program may reach during execution, the number of which is unbounded, not only the subset of states that have been tested manually. Because of composability, calls to other contracts may allow those contracts to re-enter the original token contract. When this occurs, the contract may alter either the allowance or balance associated with a particular account before earlier operations affecting that account have completed. Finally, arithmetic behavior adds a further layer of difficulty, since overflow or underflow may produce violations of balance relationships that would otherwise appear reasonable [13], although there are cases in which the observable transfer behavior of a token appears to comply with its declared interface even though the state changes associated with those transfers deviate from what actually occurred, that is, an invariant-level error [14].

Based on these gaps, this thesis focuses on invariant-based verification of fungible token standards across Ethereum and Solana.

3 Research Methodology

3.1 Research Design and Case Study Approach

This chapter outlines the methodological framework used to investigate invariant-based formal verification of token standards of Ethereum and Solana. This section of thesis discusses: the choice of token standard and implementation, the choice of formal verification tool, the criteria of invariant choice, the formal specification process, the experimental setup, and the criteria for evaluating the effectiveness of tools. Table 6, located at the end of the section, relates each research question to a corresponding methodological component supporting that research question, making the connection between design choices and research outputs explicit.

The design of the research is qualitative, artifact-driven and multi-case. For each case study, a token standard and formal invariants are chosen allowing application of verification tools that will check the chosen invariant. The case studies follow a similar structure and evaluation framework. This ensures principled cross-platform comparison, despite the fundamental differences between the Ethereum and Solana programming environments.

The idea of executing a purely quantitative was considered and rejected. Success with formal verification relies not just on quantitative metrics but also on qualities such as expressiveness, annotation burden, and what a tool can reason about. Because the two ecosystems are heterogeneous, direct numerical comparisons such as verification time would conflate tool maturity with architectural difficulty. To prevent this situation, the case study strategy applies the same invariant categories and evaluation dimensions to both of the platforms and documents the differences accordingly. Table 5 summarizes the overall research design and explains the rationale for each methodological choice.

Dimension	Approach	Rationale
Research type	Qualitative, artifact-driven	Formal verification requires specification judgment, not statistical measurement
Strategy	Multiple structured case studies	Enables systematic cross-ecosystem comparison under a common invariant framework
Token scope	ERC-20, SPL Token	Covers fungible models symmetrically across both platforms

Verification approach	Invariant-based formal verification	Invariants capture persistent correctness properties that must survive every valid state transition
Tool selection	Certora, Foundry, KEVM; Kani, Prusti, MIRAI	Tools represent distinct verification paradigms within each platform's language stack
Evaluation basis	Verifiability, expressiveness, effort, coverage, soundness of assumptions	Provides structured and consistent comparison of tool suitability across both ecosystems

Table 5: Research design overview

3.1.1 Research Question Mapping

Each research question is connected to a specific methodological component so that the study remains traceable and structured. Table 6 shows how the research questions are addressed through the selected methods and expected outputs.

Research Question	Methodological Support	Expected Output
RQ1: What functional correctness properties and security invariants define the behavior of fungible token standards?	Invariant selection criteria (Section 3.4), token standard and implementation analysis (Section 3.2)	Structured list of verified fungible token invariants per platform
RQ2: How effectively can selected formal verification tools specify and verify these properties?	Tool selection rationale (Section 3.3), verification procedure (Section 3.6), evaluation criteria (Section 3.7)	Verified / falsified / inconclusive results per tool-invariant pair
RQ3: How do architectural and execution model differences between EVM and Solana Runtime influence invariant design and verification?	Cross-platform case study design (Section 3.1), comparative invariant analysis (Section 3.4)	Comparative account of how EVM and Solana architecture shapes specification and tool choice
RQ4: What are the practical implications, limitations, and trade-offs of applying formal verification across heterogeneous blockchain platforms?	Evaluation criteria (Section 3.7), threats to validity (Section 3.8)	Practical implications and trade-off analysis for developers, auditors, and security engineers

Table 6: Research question mapping to methodology and expected outputs

3.2 Selection of Token Standards and Implementations

Two token standards form the basis of the research. The selection criteria were:

1. *Ecosystem representativeness:* This refers to the fact that these standards are real-world deployment patterns on these blockchains and not just edge cases of niche standards.
2. *Architectural contrast:* The Ethereum standards are implemented in Solidity, a contract-centric state model that targets the EVM. The Solana specifications are implemented in Rust and target an account-based runtime. My research questions center on this architectural contrast itself.
3. *Economic and practical significance:* The importance of ERC-20 and SPL is economic and practical. Their on-chain activity forms the economic majority and thus their correctness is not only theoretical but practical.
4. *Availability of stable, versioned implementations:* All four standards have well-documented, production-grade implementations available at fixed, citable versions with confirmed commit hashes.

The thesis makes use of a fixed, versioned implementation for every chosen token standard. The goal is to concentrate the verification scope on fundamental token correctness properties, thus avoiding any extensions that would add new, irrelevant state variables, account relationships, or authorization patterns.

3.2.1 Ethereum: ERC-20

The baseline implementation for ERC-20 is OpenZeppelin Contracts v5.2.0, a widely used production-grade Solidity library. The ERC-20 implementation is taken from the OpenZeppelin Contracts[1]. This version was released on 9 January 2025. Specifically, the thesis uses the base ERC20.sol contract. The following ERC-20 extensions are excluded from the primary verification scope:ERC20Burnable, ERC20Pausable, ERC20Permit.
These extensions are excluded as they add additional state variables and interaction setups that go beyond the core fungible token invariants envisioned in this thesis. Where a specific invariant relates to an extension, this is explicitly mentioned in Chapter 4. The functions of

[1] OpenZeppelin, “OpenZeppelin Contracts,” GitHub repository, commit acd4ff74de833399287ed6b31b4debf6b2b35527. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts

ERC-20 which are in verification are: *transfer, transferFrom, approve, mint, burn.* The *totalSupply* and *balanceOf* functions will be regarded as state-reading operations used inside invariant specifications, rather than independent verification targets.

3.2.2 Solana: SPL Token

The SPL Token program from the canonical solana-program/token repository is the baseline implementation for Solana fungible tokens. The repository above which was formerly maintained *solana-labs/solana-program-library* was archived on 11 March 2025 and is read-only. As such, the solana-program/token repository is the current active source of the SPL Token program. The SPL Token implementation used in this thesis is program@v9.0.0 from the solana-program/token repository published on 28 October 2025[2].
Token-2022 is explicitly excluded from the scope of this thesis. Its confidential transfer, transfer fee, interest-bearing token, and extension mechanisms introduce additional account structures and authorization patterns that fall outside the core fungible token invariants examined here. The SPL Token instructions included in the verification scope are: *InitializeMint, InitializeAccount, Transfer, MintTo, Burn, Approve.*

3.2.3 Handling Implementation Size

Some of the reference implementations include modules that are larger than suitable for direct verification with the tools chosen. Given a tool cannot successfully process a full instruction or function within some resource bound, I will extract and verify the smallest self-contained module that exercises the relevant invariant. This rule is a scope reduction rule. All state transitions and account checks on which the invariant relies on must be included in the module. The related verification chapter contains a clear exclusion statement named with a justification for any such reduction. The nature of the discipline requires any limitations to be disclosed and does not operate silently so as to dilute the verification claims.

3.3 Selection of Formal Verification Tools

The tools are selected essentially according the following three requirements: *suitable for the intended target language, implement support for invariant-based or property-based*

[2] Solana Program, “SPL Token,” GitHub repository, program@v9.0.0, commit dfb260231c761be7d9c8b63728e770a102b86495, published 28 October 2025. [Online]. Available: https://github.com/solana-program/token

reasoning, and actively maintained with good documentation available. Three tools are selected for each ecosystem.

3.3.1 Solidity tools (Ethereum)

Certora Prover: Is chosen since its CVL invariant and rule syntax map most onto the token-level correctness properties defined in Section 3.4. Thus, it is themain theorem proving tool for Ethereum in this thesis.

Foundry: Is chosen because its invariant fuzzing mode complements Certora by exploring execution sequences rather than proving properties exhaustively and its implementation in practice means results are of direct relevance for developers and auditors.

KEVM: Is chosen to exhibit a verification result based on the semantics of EVM bytecode rather than Solidity source, in order to bridge the gap between source-level proofs and what is on-chain.

3.3.2 Rust tools (Solana)

Kani: Is chosen because the bounded model checking approach is effective in verifying isolated SPL Token instruction logic at the function level when full program verification isn't feasible.

Prusti: Is chosen as it provides richer specification expressiveness than Kani through precondition and postcondition contracts, which are needed for the more complex SPL account relationship invariants.

MIRAI: Is chosen to work alongside Kani and Prusti, as it uses reasoning at the MIR level. It focuses on identifying reachability and runtime failure errors that the other two tools might miss.

3.4 Invariant Selection Criteria

The selected invariants must be relevant to token standards, security, cross-platform comparison, and formal verification. Table 7 summarizes the criteria used to decide which token correctness properties are included in the thesis.

Criterion	Description	Example
Derived from standard	The property must be explicitly or implicitly required by the token standard's specification	ERC-20: transfer operations must not alter totalSupply
Security-critical	Violation must enable a concrete attack or financial loss scenario	Unauthorized minting violates supply conservation
Cross-platform relevance	A meaningful counterpart must exist on both Ethereum and Solana	Balance consistency applies to both ERC-20 and SPL Token
Formally expressible	The property must be encodable as a precondition, postcondition, or invariant in at least one target tool	Certora invariant: transfer preserves totalSupply
Verifiable with selected tools	At least one selected tool must be capable of checking the property against the target implementation scope	Kani harness for SPL token balance update

Table 7: Invariant selection criteria

Applying these criteria yields the following invariant sets.
For fungible tokens: (1) transfer operations preserve the sender balance, recipient balance, and total supply in a consistent relationship within the modeled account set; (2) total supply may only change through authorized mint and burn operations; (3) no account balance may fall below zero; (4) delegated transfers may not exceed the approved allowance; (5) only accounts holding the mint authority may invoke minting.

3.5 Formal Specification Procedure

Each invariant is formalized by the specification language of the corresponding tool. Different tools use different notation

1. Certora: Conditions presented as CVL invariant blocks and rule declarations about contract storage variables.
2. Foundry: Features invariants that are designated in the form of Solidity functions and prefixed with *invariant_* that get executed by the fuzzer over randomized calling sequences.
3. KEVM: Refers to the specifications which are expressed as reachability claims over EVM bytecode using the reachability logic notation of the K framework.
4. Kani properties encoded as *assert!* statements inside proof harnesses with *kani::any()* symbolic inputs

5. Prusti: adds function contracts to Rust code with the *#[requires], #[ensures], and #[invariant]* macros.
6. MIRAI - properties expressed through *precondition!* and *postcondition!* macros at the boundary of a Rust function

Each specification, regardless of its tool, follows three steps. First, using plain language tied to the specific token standard to describe the vulnerability pattern it aims to protect against. Next, it gets transformed into the proper notation of the target tool, and the state variables or accounts that it evaluates are explicitly identified. Third, the formal statement is compared with the baseline implementation to ensure that it correctly reflects the property intended to be verified.

The separation of specification from verification in staged procedures is significant because if a tool proves an incorrectly stated invariant, it offers false rather than real confidence. The review step is a mandatory, not an optional process. Further, all official specifications are published with verification results to allow for an independent review.

3.6 Verification Setup and Experimental Procedure

Verification is carried out based on chosen implementations and specifications, tool by tool. For each tool-invariant pair, the procedure is to: set the tool to the target implementation and to the formal specification of the invariant, using the version and commit references given in Section 3.2; run and execute the tool and group the results as Verified: the tool shows the invariant holds in its model, Falsified: the tool produces a concrete counterexample, and Inconclusive: the tool times out, runs out of memory, or cannot process the specification; inspect any counterexamples for genuine implementation errors, specification errors, or any artifact of the toolkit's modeling assumption; and capture the outcome with tool name and version, invariant identifier, implementation file and function scope, result category and any relevant tool output or error message.

Verification of the *OpenZeppelin v5.2.0* implementation’s Solidity source targets the Ethereum tools, with KEVM running on the compiled bytecode. Solana tools target Rust source of the *SPL Token program@v9.0.0* modules identified in Section 3.2. If scope is reduced to representative module, this is described in experiment beforehand. All experiments are fully documented to allow reproducibility and provide the traceable evidence base for Chapter 6.

3.7 Evaluation Criteria for Tool Effectiveness

The tool's effectiveness is assessed across five dimensions of Verifiability, Expressiveness, Specification effort, Coverage and Soundness of assumptions . I score each dimension on a three-level scale of *fully supported, partially supported, and not supported*. A three-level categorical scale is used instead of a numerical score because the judgments involved are qualitative and the categorical scale is more honest about the level of precision available than a finer-grained numeric one.

Verifiability measures whether the tool produced a conclusive result, either verified or falsified, rather than an inconclusive one. It is rated fully supported when a definite result is achieved, partially supported when a result is obtained only for a reduced or simplified scope, and not supported when the tool is unable to process the specification or crashes.

Expressiveness measures how easily the invariant can be written in the tool's specific language. It is rated fully supported when the invariant can be described completely without omitting any part of the specification, partially supported when some aspects have to be omitted, and not supported when the invariant cannot be described in a way that yields meaningful results.

Specification effort measures the annotations, harness, and setup required to implement the task. It is rated fully supported when only minimal setup and the invariant statement are needed, partially supported when moderate setup such as harness construction or instrumentation is required, and not supported when the setup burden is too great to test the specific invariant.

Coverage measures the execution paths and account configurations that the tool is able to reason about. It is rated fully supported when the tool reasons over all states within the specified model, partially supported when the tool covers only a subset of those states, and not supported when the tool does not cover enough states to provide meaningful results.

Soundness of assumptions concerns how far the verification results depend on the assumptions made about the runtime environment, account structures, functions, and tool calls. This is particularly critical for Solana, where the account-based runtime environment must be modeled or abstracted. It is rated fully supported when verification holds under a runtime almost identical to the actual one, partially supported when verification holds under

simplified but plausible assumptions, and not supported when the assumptions required are very different from the actual runtime environment and behavior.

3.8 Threats to Validity

This section sets out the threats that could limit the validity of the study and the measures taken to contain each one. Following established guidance for case study research in software engineering, I group them under internal, construct, and external validity, together with threats specific to formal verification such as harness modeling and tool limitations [65].

Specification quality (internal validity): When an invariant is improperly formalized, a verification result indicates a specification error and not that of the implementation [60]. The three-step specification process outlined in Section 3.5, together with the documentation of all formal specifications next to the results for independent examination helps in mitigation.

Cross-ecosystem equivalence: The ERC-20 and SPL Token are similar in functionality but not identical in design. SPL Token uses separate mint and token accounts where ERC-20 uses contract-internal mappings, and the authorization models differ structurally [17]. Consequently, invariant comparison across platforms is a comparison of analogous, not identical, properties.

Harness and modeling threat: Kani verification harnesses and KEVM specification models abstract away from the full blockchain runtime in a necessary manner; gas mechanics, account rent, cross-program invocation behavior, and Solana's parallel execution scheduler [27]. The verification model might overlook behaviors that can actually be achieved in a real deployment due to these abstractions [24]. Every harness and its assumptions are explicitly documented in order to bound this threat.

Tool limitation threat: The tools may not currently have the capacity to enable full program verification for Rust and Solana-specific tools [32]. Kani's bounded model-checking cannot be adapted for unbounded loops [27]. When limitations of the tool force a reduction of scope, this is logged and considered in the tool's verifiability and coverage ratings.

Researcher interpretation threat: The evaluation dimensions mentioned in Section 3.7, especially expressiveness and specification effort, are subjective [66]. Various experts might evaluate the output of the same tool differently. To partially mitigate this, the three levels are

accompanied by concrete descriptions of each level, and the raw tool output is preserved to allow for an independent re-evaluation.

Construct validity: The five evaluation dimensions might not actually reflect every aspect of tool usefulness to practitioners. While selecting all critical dimensions in the formal verification literature and in the practical concerns of blockchain security auditors partially helps, one cannot rule out on missing some important dimensions.

External validity: The results might not apply to other token standards, implementations and tools. This thesis does not intend to assess all token implementations and all verifying tools. The reasons for selection in Sections 3.2 and 3.3 indicate that the selected cases are intended to be representative or chosen token and blockchains

4 Fungible Token Correctness Properties and Invariants

4.1 Fungible Token Correctness Properties and Invariants

Token fungibility refers to the fact that every unit of one kind of token has the same inherent value. Every unit of the same token is therefore identical and can be exchanged with another unit of the same token. Therefore, every token balance can also be made up of units of the same kind of token. This property is central to how token standards such as ERC-20 and SPL Token are designed, as both are built around the assumption that all units of a given token are interchangeable [13], [17].

There are several ways in which accounts interact with fungible tokens e.g., *transferring tokens between accounts, minting new tokens into an account, burning tokens out of an account*. As such, there are certain required properties that should always hold true regarding the relationships *between token balances, total supply of tokens, transfers of tokens, minting/burning* rules and delegation. Loporchio et al. observe that ERC-20 tokens exhibit structurally different transfer patterns than other standards, which underlines that the correctness properties of fungible tokens must be studied as a distinct category [6].

These required properties will serve as the basis for demonstrating whether or not fungible token transitions produce correct results. This chapter defines what is meant by *"correct"* for fungible tokens. Specifically, *"correct"* means that every transition from a valid initial state to a final state produces a state where the arithmetical properties such as those related to how many tokens exist and authorization properties such as who can create/move tokens of the token system are preserved. Bhargavan et al. describe this goal as reasoning about both runtime safety and functional correctness of contract behavior, arguing that formal verification is necessary because testing alone cannot guarantee that properties hold across all reachable states [7]. Thus, a successful transfer of tokens should result in the movement of some number of units from the source account to the destination account without creating or deleting any additional tokens. Similarly, a successful minting action should increase the total number of existing tokens by the number being minted; however, only when performed by an authorized party. Likewise, a successful burn action should decrease the total number of tokens in existence by the number burned; however, only when performed via a valid burn path. Finally, a successful delegated transfer should honor any allowance or delegate authority limitations. Failure to satisfy any one of these requirements could cause the token system to

enter an invalid state resulting in potential inflation, lost funds, double-spending, or simply poor accounting practices. Atzei et al. provide a taxonomy of Ethereum smart contract vulnerabilities that demonstrates how implementation mistakes in exactly these areas; authorization, balance updates, and supply control can lead to exploitable contract behavior [15], [22].

While the properties described above are generally applicable to fungible tokens on either platform and thus represent conceptual property requirements for both ERC-20 and SPL Token models, they need to be defined differently due to differences in how each platform represents token state. Ethereum stores token state inside contract storage mappings accessed through the EVM [4], [11], while Solana represents token state through separate mint accounts, token accounts, and authority fields that programs operate over at runtime [17], [18]. Section 4.2 describes how the general properties defined in this section are specifically refined for use in relation to the ERC-20 model; similarly, Section 4.3 describes how the general properties defined in this section are refined for use in relation to the SPL Token model.

4.1.1 Balance Consistency

The overall goal of balance consistency is to ensure that when we account for each individual user's balance in relation to their respective holdings of the given token and total supply of the token that these two will always be balanced regardless of what valid changes occur to the system. When we update users' balances and/or issue new tokens without performing some other valid action on the system, the internal logic of our token accounting model breaks down.

Given a group of token holding accounts A, a function describing each account's current balance b(a), and the total supply of the token S, the invariant related to balance consistency is defined as:

$$\sum b(a) = S, \text{ for all } a \in A$$

This relationship is implicitly required by the *totalSupply* and *balanceOf* interface definitions of the ERC-20 standard [13] and holds equivalently for SPL Token across all initialized token accounts associated with a given mint.

In general terms, A would represent the complete set of all unique addresses in existence. However, since tools used for verification cannot examine every single address in existence nor can they consider all potential token accounts at once [24], we verify this condition through an inductive approach. An inductive argument involves demonstrating the truth of a statement by showing that if it is true prior to a state transition occurring, and only those accounts being modified were done so by changing their values by exactly the amount specified by the operation itself, then the statement remains true after the operation has completed [7]. With respect to valid transfers as described in Section 4.2, both the sender's account balance is decreased, and the recipient's account balance is increased by the exact same amount resulting in no net change to the total. Valid minting operations increase supply by the same amount credited to the receiving account. Lastly, Valid burning operations decrease supply by the same amount debited from the source account. As such, the specifications included in Sections 4.2 and 4.3 encapsulate this inductive proof into local pre and postconditions rather than one large universal quantifier [28]. When an invariant related to balance consistency is violated, it implies that there are tokens being created or destroyed outside of a valid minting or burning operation. Therefore, balance consistency is identified as the primary invariant within this chapter and is also the central component upon which all tool specifications are ultimately based.

4.1.2 Supply Preservation and Minting Constraints

Supply preservation means transfers cannot affect the number of total tokens within circulation. Thus, after each transfer, the new total supply will equal the original total supply:

$$S\_after = S\_before$$

The supply can be changed with special *supply-changing actions* like *minting* (adding tokens to supply) and *burning* (removing tokens from supply), but these are governed by *authorization rules* for the platforms they run on [13], [17]. *Authorized entities* can *mint*, increasing the *total supply*, as well as *burn*, decreasing the *total supply*. In the case of minting, only authorized parties can create additional supply. The OpenZeppelin ERC-20 base contract mints internally and allows deployment contracts to expose the *authorization mechanism* [25] for minting. The SPL Token also controls who can mint, based on the mint_authority value found in the *mint account* [17]. *Burning* authorizations differ in ERC-20 vs SPL Token. ERC-20 burns depend on how the deployer implemented exposing burn functionality. SPL Tokens

allow burning to occur either by the *token account owner* or one of their valid *approved delegates* [22].

The general *supply invariant* consists of three aspects: all transfers will leave *total supply* unchanged; only those who are permitted to perform *mint operations* may add to the total supply; only those who are permitted to perform *burn operations* may remove from total supply. This is very important because *unauthorized minting* increases supply, thereby devaluing current owners' holdings, and *unauthorized burning* eliminates user funds [15]. Ivanov et al. documented instances of unauthorized changes to *supply-changing functions* resulting in potentially valuable tokens being inflated and creating exploitable vulnerabilities [30], which is why this invariant must be upheld in production-ready token systems.

4.1.3 Transfer Validity

Validity for transfers defines the arithmetic result of transferring tokens from account *s* to account *r*. All transfers of token amount *v* between accounts *s* and *r* must meet these three *validity conditions* at once.

$$b(s)\_after = b(s)\_before - v$$

$$b(r)\_after = b(r)\_before + v$$

$$S\_after = S\_before$$

Before executing a transfer, the sender must also have a balance of at least *v*.

$$b(s)\_before \geq v$$

If the sender's balance is less than *v* when he attempts to execute a transaction, the operation will either fail (and possibly roll back) rather than create an inconsistent state and thus prevent the creation of *negative balances*; prevent *arithmetic underflows*; and prevent *double spending [4]*, [15]. The sender's balance update cannot be validated alone as the recipient's balance update cannot be validated alone. If you decrease the sender without increasing the recipient, you destroy tokens. If you increase the recipient without decreasing the sender, you create tokens. Both ways cause both violations of *balance consistency* and violations of *supply conservation* at once [7], [20].

In addition to the validation requirements above, a valid transfer must preserve *frames*. This means that a valid transfer should not change any other balance or supply related state outside

of what was changed due to the transfer [28]. *Frame preservation* is important here since there could exist some implementations which validate the sender and recipient locally but then also change incorrect unrelated accounts. Specifications for validation described in Chapter 5 will implicitly describe the preserved frames through bounding the number of states that can be modified by functions during their operations.

4.1.4 Delegation and Authorization Rules

Fungible Tokens can perform *authorized transfers*. An account can give permission to another to perform token transfers on its behalf up to a specified maximum. In ERC-20, the *allowance* is implemented using a mapping called *_allowances* from an *Owner* to a *Spender* [13], while SPL Token implements *delegation* via a *Delegate* field and a *Delegated Amount* field in each Token Account [17]. The primary *authorization invariant* in both implementations is that no account shall be allowed to perform a transfer of tokens from another account unless they are the *Owner* of that account or hold the proper level of *delegated authority* [22].

In the case of a delegated transaction of amount V:

$$Allowance(Owner, Spender) \geq V$$

Following a successfully completed delegated transaction, the allowance will have decreased by the exact amount of the transferred amount:

$$Allowance\_after = Allowance\_before - V$$

The above rules must be interpreted within the context of the implementation under review. For example, when using OpenZeppelin's ERC-20 implementation, a spender approved with a uint256 maximum value is treated as having been granted *infinite approval*. The OpenZeppelin implementation leaves the allowance unchanged after executing *transferFrom* [25]. Therefore, the specification must clearly differentiate between normal *finite approvals* and this special *infinite approval* case. Failing to make this differentiation would result in a correct implementation appearing to violate the postcondition of reducing the allowance. Perez and Livshits report that *allowance misuse* is among the top five most frequently abused exploits found in deployed ERC-20 smart contracts [31].

An owner may grant a spender approval for an amount larger than their current balance at the point of approval. This alone does not constitute an error. What constitutes an error is when the owner's balance is less than what was approved for at the time the transfer is executed

[13]. As stated previously there are three conditions for the *delegated transfer invariant* to be satisfied: the spender must be approved to act on behalf of the owner; the allowance must be sufficient at the time the transfer is executed; and the allowance must be adjusted accordingly following a successful completion of the delegated transfer, subject to the exception of an *infinite approval*. Should either of these conditions not exist then an unauthorized entity could either move funds or reuse the same authorization on subsequent transactions [15], [31].

The four properties defined in this section together constitute the *invariant framework* applied to ERC-20 in Section 4.2 and to SPL Token in Section 4.3.

4.2 ERC-20 Correctness Properties and Invariants

The ERC-20 protocol describes an established interface for fungible tokens on the blockchain platform of Ethereum and has been the de facto standard for token creation since it was proposed by Vogelsteller and Buterin in 2015 [13]. Therefore, when there are errors within ERC-20 implementations (i.e., incorrect implementations), these errors have far reached effects economically, as there are billions of dollars in value being stored and moved via ERC-20 tokens every day [6]. Large-scale Ethereum token datasets further demonstrate the economic and empirical relevance of ERC-20 transaction activity [23]. This section outlines how the general invariant structure described in Section 4.1 will be adapted to fit the specific ERC-20 implementation utilized throughout this thesis: OpenZeppelin Contracts version 5.2.0, pinned to commit acd4ff74, utilizing the base ERC20.sol contract [25], as it is the most commonly implemented production quality library for Solidity code and the vast majority of ERC-20 contracts utilize OpenZeppelin Contracts either directly or indirectly [25].
In regard to the use of OpenZeppelin Contracts, the base ERC20.sol contract does not provide any form of enforcement for mint authorization. The _mint function is defined as internal; therefore, any form of authorization logic must be defined by the implementing contract. This choice by OpenZeppelin was intended to allow developers to define their own authorization logic for minting new tokens, however, this choice creates a disconnect between what the standard expects in terms of ensuring correct operation versus what the base contract itself can guarantee regarding correctness [25]. These limitations are expressed in the invariant set provided below and expanded upon in the invariant set below.

4.2.1 ERC-20 Storage Layout

ERC-20 token state in the OpenZeppelin implementation is stored entirely within the contract's internal storage through three variables:

1. *_balances*: mapping of an address to a *uint256* value; maps each account's balance.
2. *_allowances*: a nested mapping from owner address to spender address to *uint256*, storing delegated spending limits
3. *_totalSupply*: a *uint256* scalar storing the total number of tokens in circulation

All correctness *invariant* conditions for *ERC-20* are expressions on *constraints* over the three *storage variables* and the *functions* that read/write them. As noted by Atzei et al., many *vulnerabilities* arising from incorrect updates to *storage variables* of this kind are due to an incorrect update being made to one variable while another related variable is left *inconsistent* [15]. Figure 4 visualizes these dependencies, mapping the three storage variables _balances, _allowances, and _totalSupply to the transfer, approval, mint, and burn functions that read and modify them.

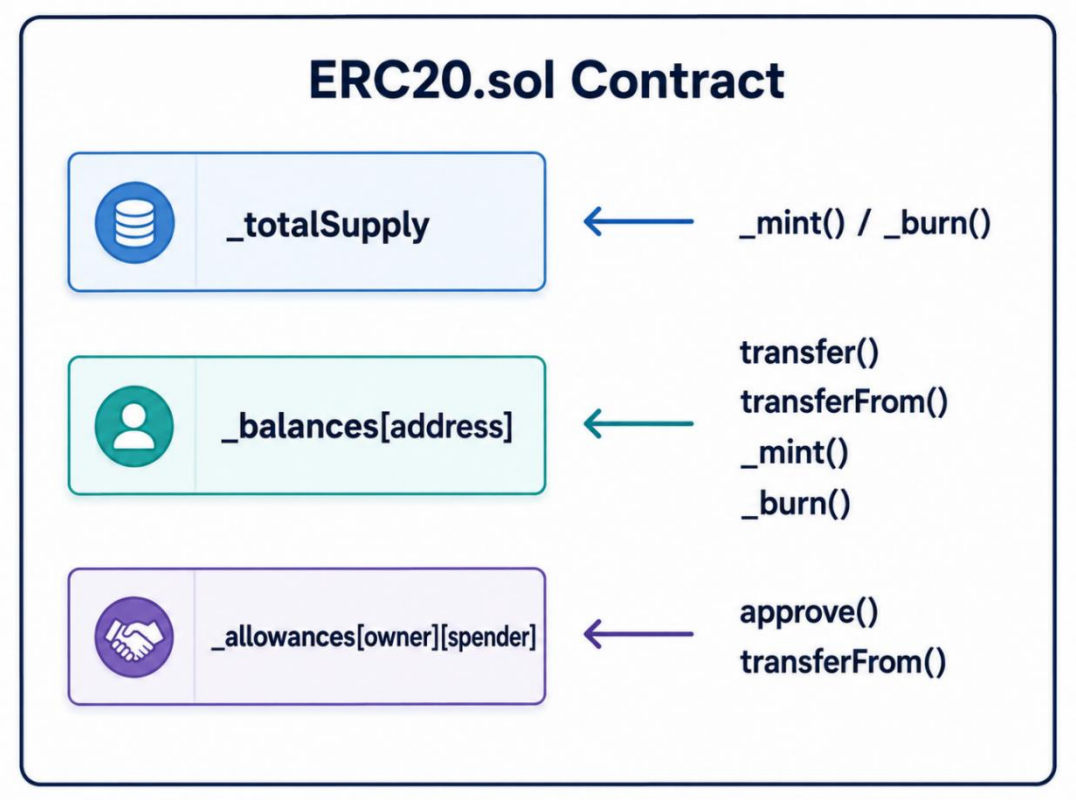

Figure 4: Relationship between ERC-20 storage variables and the functions that modify them

4.2.2 Functions in Verification Scope

The following five functions from ERC20.sol will be verified in this chapter. The totalSupply and balanceOf functions are *read-only state-extracting operations* that can be used within an *invariant specification* and thus do not represent a separate target of independent verification. A description of the complete function signatures and their documented usage is available in the OpenZeppelin ERC20.sol source code and developer documentation [25] and, therefore, the expected semantics of these functions is described by the *EIP-20 standard* [13].

1. transfer (address to, uint256 amount): transfers tokens from the caller's account to a recipient; modifies _balances only [13], [25]

2. transferFrom (address from, address to, uint256 amount): transfers tokens from a specified account using an approved allowance; modifies _balances and _allowances [13], [25]

3. approve (address spender, uint256 amount): sets the allowance a spender may use on behalf of the caller; modifies _allowances only [13]

4. _mint (address account, uint256 amount): *internal function* that creates new tokens and credits them to an account; modifies _balances and _totalSupply [25]

5. _burn(address account, uint256 amount): *internal function* that destroys tokens from an account; modifies _balances and _totalSupply [25]

The most important observation from this evaluation is that the transferFrom operation is the only operation that affects two *storage objects* simultaneously, i.e., _balances AND _allowances, thereby making it the most difficult to prove correct. The reason for this complexity is that there are two *storage variables* that need to be updated at the same time during the execution of one single function [15], [24]. As a result, Hildenbrandt et al. state that this *simultaneous update requirement* represents the major difficulty in verifying ERC-20 bytecode through use of KEVM at the *bytecode level* [24]. Therefore, certification of its behavior at the *source code level* does not provide sufficient assurance. According to Bhargavan et al., proof of correctness at the source code level provides no assurances regarding preservation of the same properties in the *compiled bytecode* [7].

4.2.3 ERC-20 Specific Invariants

Applying the general invariant framework from Section 4.1 to the ERC-20 storage model and the five functions above yields six concrete invariants listed in Table 8.

Invariant ID	Description	Storage Variables	Functions in Scope
ERC20-I1	Sum of all _balances entries equals _totalSupply	_balances, _totalSupply	transfer, transferFrom, _mint, _burn
ERC20-I2	No _balances entry may underflow below zero	_balances	transfer, transferFrom, _burn
ERC20-I3	transfer and transferFrom do not change _totalSupply	_totalSupply	transfer, transferFrom
ERC20-I4	transferFrom may not exceed the approved _allowances entry	_allowances, _balances	transferFrom
ERC20-I5	Only _mint and _burn may change _totalSupply	_totalSupply	All public functions
ERC20-I6	Only an authorized caller may invoke _mint	_totalSupply, _balances	_mint

Table 8: ERC-20 invariants mapped to storage variables and functions in scope

ERC20-I1: Balance Consistency

ERC20-I1 is an application of the balance consistency property for the generalized form described in Section 4.1.1 for the ERC-20 token standard. The total of all balances in the mapping _balances must at all times be equal to _totalSupply. Each time _mint is called it will increase the specific entry in _balances by a value identical to what has increased _totalSupply. Likewise, each time _burn is called, it will decrease _totalSupply by an amount identical to what decreased a specific _balances entry. Finally, each time either transfer or transferFrom is called, one specific _balances entry will be decreased while another will be increased by the same amount; thus, no change will occur with respect to _totalSupply. Violation of ERC20-I1 could indicate creation or destruction of tokens through some path other than those provided in this contract for such actions, which according to Atzei et al. represents one of the two types of financial failures possible with a smart contract; that being creation or loss of assets [15].

ERC20-I2: Non-Negative Balances

No values can exist in _balances less than zero. The arithmetic underflow condition on non-negative unsigned integers has been given automatic revert capability since the introduction of Solidity version 0.8. As such, for all unsigned non-negative integer data types, ERC20-I2 is enforced by the programming language. It still remains as a formal verification objective because one must first prove the revert will occur as expected before applying the revert and changing state [4].

ERC20-I3: Transfer Supply Conservation

Neither transfer nor transferFrom can be used to update _totalSupply. This is part of the ERC-20 refinement on the supply conservation property described in Section 4.1.2. In the OpenZeppelin version of these two functions, each function will modify some _balances entries but will not modify the _totalSupply variable itself. The verification process needs to establish that there are no indirect ways for either function to reach _totalSupply via other functions. This is difficult at the bytecode level as the EVM does not enforce any language-level access control over storage variables, which makes the relevance of KEVM verification especially strong for this invariant [13], [24].

ERC20-I4: Allowance Bound on transferFrom

transferFrom can move an amount from msg.sender no greater than the current allowance stored in _allowances[from][msg.sender]. After a transaction has been successfully completed the allowance entry should have been reduced by the exact amount moved. For the sake of completeness, it is worth noting that as discussed in Section 4.1.4, the OpenZeppelin implementation considers a uint256 max approval as unlimited and therefore will leave _allowances untouched when such a scenario is encountered.

Therefore, an appropriate way to specify the invariants is as follows:

1. For finite approvals: _allowances[from][spender]_after = _allowances[from][spender]_before − amount
2. For infinite approvals: _allowances[from][spender]_after = _allowances[from][spender]_before

Perez and Livshits have stated that one of the top three most common exploitation methods for deployed ERC-20 contracts was allowance misuse, further noting that the gap between the approval state at the time a user gives their approval and the actual token balance available at the time the transaction executes is a very commonly exploited condition [31].

ERC20-I5: Supply Modified Only by Mint and Burn

No function other than _mint and _burn may change _totalSupply. This is verified across all public functions of the contract simultaneously using a universal method quantifier in the Certora specification in Section 5.1.1. It is the strongest supply conservation invariant because it does not restrict itself to transfer functions but covers every callable function in the contract [7]. Bhargavan et al. argue that this class of global property is precisely where formal verification outperforms testing, since a test suite cannot enumerate every possible function call sequence [7].

ERC20-I6: Mint Authorization

Only a call from an approved caller can use the supply-increasing pathway through _mint. The ERC20.sol contract in OpenZeppelin has declared _mint as internal with no authorizations of its own [25]. Therefore, all authorization for _mint will depend on the calling contract, which means that ERC20-I6 cannot be completely verified based solely upon the base contract. Smolka et al. note that one of the major error-prone areas found among both Ethereum and Solana smart contracts are missing authorizations to privileged actions [8], and therefore, ERC20-I6 will have to be reviewed at the level of how the contract was deployed, regardless of whether or not the base contract is able to enforce it.

4.2.4 Interaction Between ERC-20 Invariants

The six invariants do not operate in isolation. *ERC20-I5* provides a comprehensive base for all other invariants including the enforcement of boundary conditions for *ERC20-I3* and *ERC20-I6*. In addition, *ERC20-I1* relies upon both *ERC20-I2* and *ERC20-I3* being true at the same time. If *balances* "under flow" (i.e., are too small) or if any single *transfer* alters *total supply,* then this causes a break in *balance consistency*. The reason *ERC20-I4* interacts with *ERC20-I1* is due to the fact that when an *allowance* violation occurs which would allow some arbitrary *transfer* of funds it will also create a false state regarding the *total balance*. This means that a failure in any one of the six *invariants* listed above will result in a failure in at least one of the remaining five *invariants*; therefore, verifying just part of the *invariants* is not

sufficient; all six must be verified collectively in order to confirm that the *ERC-20* model is valid [7], [15]. These invariants are therefore based on the state of each component of the allowance system, and so they cannot fully represent every correctness property of that system. The well-known approve race condition, for example, in which an approved spender front-runs an owner's transaction that changes a non-zero allowance to a new non-zero value, is an order of transactions that a single before-and-after state relation cannot describe.

4.3 SPL Token Correctness Properties and Invariants

The *SPL Token Program* is the canonical *fungible token* standard for the Solana Blockchain. It differs significantly from *ERC-20*; whereas ERC-20 stores all token state inside one *Smart Contract* (i.e., there is only one account), the SPL Token program uses an explicit pass of accounts related to tokens, with their relationships being validated at runtime [17], [18]. As such, it has significant implications for the specification and verification of *correctness properties* as well. In this section, we will refine the general *invariant framework* of Section 4.1, specifically for the SPL Token Program version @v9.0.0 (as pinned in commit dfb260231) used in this thesis: SPL Token Program@v9.0.0, referenced from Github and located in the solana-program/token repository.

One of the most important distinctions between *SPL Token* and *ERC-20* is that the SPL Token program is completely *stateless*. Therefore, it does not contain any token balances or supply information. All token state is spread out over different *on-chain accounts*. Each time an operation (*instruction*) is executed, these accounts need to be provided explicitly. Therefore, correctness in SPL Token is determined by two factors: whether the *program logic* is correct and if the correct accounts were passed into the program (with respect to the owner of those accounts), properly initialized, and signed by the proper *authority* [17], [22]. Cui et al. demonstrated using VRust that the most critical weakness types in Solana programs include failing to check *ownership*, failing to verify *signers*, and having incorrect *account relationships*; all of which correspond to violations of an *invariant* in the account-based model for SPL Tokens [32].

4.3.1 SPL Token Account Layout

The state of the SPL Token will exist across two primary account types: the mint account as well as the token account. To create an understanding of how to define invariants (i.e., constraints on data), it is first necessary to understand the general composition of each

account type. Invariant definitions are ultimately based upon field(s) from one or both of the above-mentioned account types. The relationship between accounts for this particular example is illustrated by Figure 5.

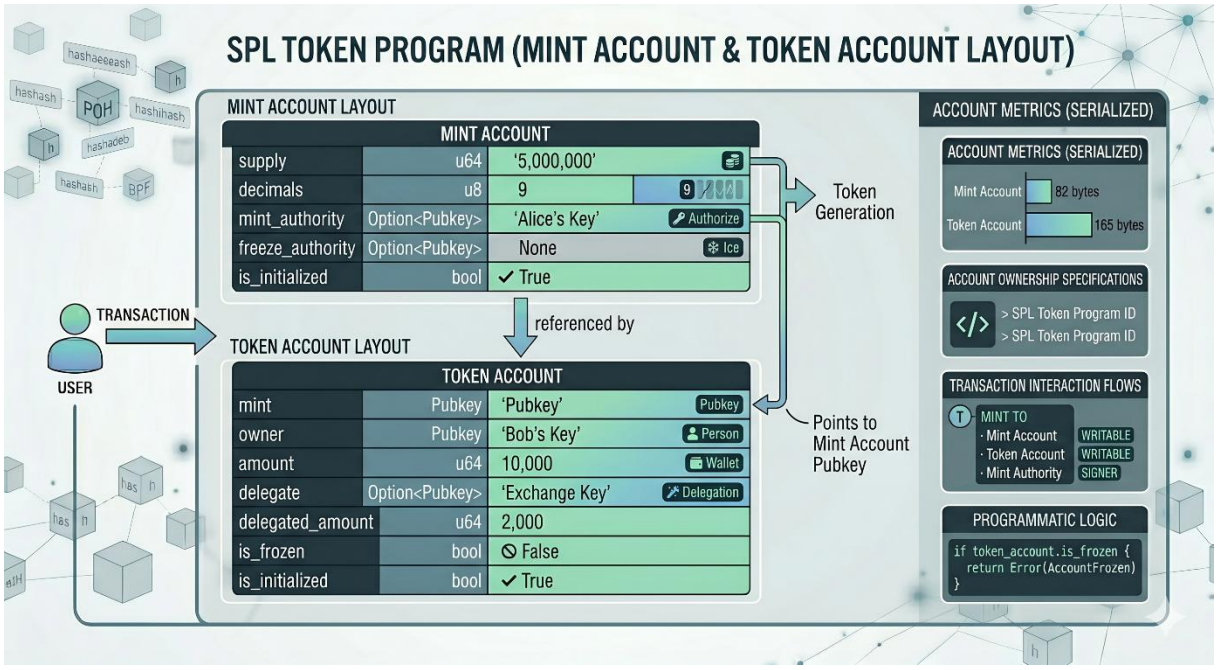

Figure 5: SPL Token mint account and token account field layout and their relationship

The *mint* field in every *token account* contains the public key of the *mint account* associated with it. These two are connected and form the base for multiple invariants; specifically, a token account can never reference an incorrect mint and all modifications to the total supply will have to go through the mint referenced by the token account [17], [35]. Navas demonstrated in the formal verification of SPL Token 2022 that the connection between the *mint* and its *token accounts* forms the primary *structural constraint* on which all proofs related to supply will rely [33].

4.3.2 Instructions in Verification Scope

The following six *SPL Token Program* @v9.0.0 instructions below are in scope for verification within this chapter. In Solana, *instructions* are like Solidity functions; however, instead of referencing an internally stored *contract state*, each instruction receives its state via explicit references to accounts [18].

1. *InitializeMint*: creates and initializes a new *mint account* with a specified decimal precision, *mint authority*, and optional *freeze authority*; writes to the mint account [17]

2. *InitializeAccount*: creates and initializes a new *token account* associated with a specific mint and owner; writes to the token account [17]
3. *Transfer*: moves a specified token amount from one token account to another; modifies *amount* fields in both source and destination token accounts [17], [35]
4. *MintTo*: mints new tokens into a token account; modifies *amount* in the destination token account and *supply* in the mint account; requires valid *mint_authority* [17]
5. *Burn*: destroys tokens from a token account; modifies *amount* in the source token account and *supply* in the mint account [17]
6. *Approve*: sets a *delegate* and *delegated amount* on a token account, authorizing a third party to transfer up to the specified amount [17]

A very important structural issue is that both *MintTo* and *Burn* update fields in two completely different accounts at once; the *amount* field in the *token account* and the *supply* field in the *mint account*. Therefore, SPL Tokens have the same *dual-update problem* as ERC-20's *transferFrom* described in Section 4.2.2. The architectural complexity however exists since the two accounts being updated are passed as individual parameters to the instruction and therefore must first validate that they belong to the same mint before any updates can occur [32], [33]. Certora's formal verification work on SPL Token 2022 confirms that encoding this *cross-account relationship* as a precondition is the most critical specification challenge for supply-related invariants [36].

4.3.3 SPL Token Specific Invariants

The use of a generalized form of an invariant as described in Section 4.1 on the SPL token Account model results in six specific invariants. As well as differing in what field these constraints are applied against (i.e., which field the constraint is against), there are other relationships among accounts that do not exist for an ERC-20. There are other account relationship invariants that do not directly have an equivalent ERC-20 constraint. All of these can be found in Table 9 below.

Invariant ID	Description	Account Fields	Instructions in Scope
SPL-I1	Sum of all token account amounts for a mint equals mint.supply	mint.supply, token_account.amount	MintTo, Burn, Transfer
SPL-I2	No token account amount may underflow below zero	token_account.amount	Transfer, Burn
SPL-I3	Transfer does not change mint.supply	mint.supply	Transfer
SPL-I4	Only the mint authority may invoke MintTo	mint.mint_authority	MintTo
SPL-I5	A token account must be owned by the expected mint	token_account.mint, mint pubkey	Transfer, MintTo, Burn
SPL-I6	Approved delegate cannot transfer more than the delegated amount	token_account.delegate, delegated_amount	`Transfer

Table 9: SPL Token invariants mapped to account fields and instructions in scope

SPL-I1: Balance Consistency

SPL-I1 is the SPL Token version of balance *consistency* invariant from Section 4.1.1. The total of all amounts stored on all *token accounts* owned by a mint will always equal the *supply* number of that mint. *MintTo* changes the *amount* for each involved token account and adds the same value to *mint.supply*. *Burn* changes both the involved token accounts' *amount* and *mint.supply* by the same amount. *Transfer* changes the *amount* on two token accounts but does so in opposing directions and therefore leaves *mint.supply* unchanged. In contrast to ERC-20, where it can be established using one contract's mapping, SPL-I1 needs to establish this relationship between different accounts, as they have no direct reference to each other [33], [36].

SPL-I2: Non-Negative Amounts

No *token account* balance will ever go below zero. The same enforcement mechanism used in ERC20-I2 uses Rust's type system via *u64* to prevent *arithmetic underflows* from causing panic or revert at runtime [5]. However, since we want to formally verify the panic occurring before any state changes are made, we still consider this an additional *formal verification* target. Cui et al. describe *arithmetic overflows* and *underflows* as one of the top eight types of

vulnerabilities in Solana programs, pointing out that *checked arithmetic* needs to be explicitly used on release builds of Rust, since overflow checking is disabled by default [32].

SPL-I3: Transfer Supply Conservation

Transfer cannot alter *mint.supply*. This is an implementation of the *supply conservation* property from Section 4.1.2 in the SPL Token program. Within the SPL Token program, *Transfer* receives the *mint account* as input solely to validate the action and therefore cannot modify it. The verification process will ensure there are no paths through *Transfer* that reach the *supply* field of the mint account. The benefit of structurally enforcing this invariant over its ERC-20 variant is due to the fact that when *Transfer* receives the mint account as a *read-only reference*, Rust's *ownership model* restricts all writes to it at the language level; additionally, these restrictions may be verified using the Kani and Prusti verification tools in Section 5.2 [32], [34].

SPL-I4: Mint Authority Enforcement

Only the account that has been assigned to *mint_authority* can successfully call *MintTo* for the mint account. As discussed in ERC20-I6, the base OpenZeppelin ERC-20 contract has no analogous relationship and delegates all authorization to the contract that deployed it. The *mint_authority* is directly stored within the *mint account* and must be checked by the program prior to allowing any increases to total supply [17]. Cui et al. found *missing signer checks* and *missing owner checks* to be the two most common forms of critical vulnerability class across Solana programs, both of which have direct implications on SPL-I4 violations [32].

SPL-I5: Token Account Mint Ownership

The *token account's mint* field must match that of the *mint account* provided when a transaction modifies a token account's supply or passes it in a transfer. There is no explicit equivalent in ERC-20 tokens, since all token state exists in a single contract and relationships between accounts are implicit by default. However, with external passing of accounts in SPL Tokens, an attacker could simply provide a token account with a different mint. Therefore, the program will need to validate the *mint* field on every token account against the mint's expected *public key* before allowing any changes to either the balance or supply [22], [32]. According to the Solana Foundation's security guidance, checking the *account relationship* between the token account and mint is required of any valid SPL Token program [22].

SPL-I6: Delegate Amount Enforcement

A *delegated transfer* may only be for an amount less than or equal to the amount in the *delegated_amount* field of the sender's token account. The amount successfully transferred via a delegated transfer will reduce the *delegated_amount* field of the sender's token account by that same amount. This is equivalent to ERC20-I4 but has a key structural difference: it stores the *delegated amount* per *token account* rather than per user and token type, which allows the invariant to express itself over a single account's attributes rather than across two different accounts [17]. Perez and Livshits identified *delegated transfers* as one of the top three most vulnerable attack vectors in fungible tokens, and this vulnerability also exists with respect to SPL Token's *delegation mechanism* [31].

4.3.4 Comparison with ERC-20 Invariants

The six SPL Token invariants described in Section 4.3 have an identical conceptual correspondence as do the six ERC-20 invariants described in Section 4.2.3. However, there are two significant differences when considering the concrete representation of these concepts. The first difference lies in how explicitly SPL Token invariants express account relationships, specifically via SPL-I5, where ERC-20 invariants have no such equivalent as they only concern contract-internal state. Secondly, the inductive argument to prove SPL-I1 must be constructed across many separate accounts versus the single mapping used by ERC-20; this makes it more difficult to represent in formal verification tools [33], [36]. Table 10 summarizes the direct correlation between both sets of invariants.

ERC-20 Invariant	SPL Token Invariant	Key Structural Difference
ERC20-I1: Balance consistency	SPL-I1: Balance consistency	SPL spans multiple accounts: ERC-20 uses one mapping
ERC20-I2: Non-negative balances	SPL-I2: Non-negative amounts	Both enforced by language type system
ERC20-I3: Transfer supply conservation	SPL-I3: Transfer supply conservation	SPL uses read-only mint reference; ERC-20 has no such enforcement
ERC20-I4: Allowance bound	SPL-I6: Delegate amount bound	SPL stores per-account; ERC-20 uses nested mapping

ERC20-I5: Only mint/burn change supply	SPL-I4: Mint authority enforcement	SPL stores authority in mint account; ERC-20 delegates to deployer
ERC20-I6: Mint authorization	SPL-I5: Token account mint ownership	SPL requires explicit account validation; no ERC-20 equivalent

Table 10: Comparison of Invariants

5 Results and Analysis

The results of this chapter present a series of verifications for the ERC-20 and SPL Token invariants defined in Chapter 4. The process employs both the verification procedure and the outcome categories defined in Section 3.6. Section 5.1 addresses the Solidity-based verification of the ERC-20 invariants using Certora, Foundry, and KEVM against the OpenZeppelin ERC20.sol implementation. Section 5.2 provides the Rust-based verification of the SPL Token invariants using Kani, Prusti, and MIRAI against the account models reconstructed from Section 4.3.1. Section 5.3 synthesises these results under the five criteria presented in Section 3.7.

5.1 Solidity-Based Verification

This part of the thesis describes the formal validation of the *ERC-20 invariants* defined in Section 4.2.3 using three *Solidity*-oriented formal verification tools: *Foundry*, *Certora Prover* and *KEVM*. Each of these tools follows one of the paradigms for *formal verification* identified in Section 3.3.1. The *Certora Prover* provides a *deductive* rule-based verification against the *Solidity* source code. The *Foundry* executes random call sequences to perform *invariant fuzzing*. The *KEVM* validates an *EVM* binary file through *reachability reasoning*. Taken together, they offer a complete view over all the verification dimension as defined in Section 3.7. This part also responds directly to RQ2: How effective are some formal verification tools to define correct properties in Section 4.2.3 and to formally prove their correctness? Each specification was developed according to the formal specification process proposed in Section 3.5. First, we describe each *invariant* in natural language related to the token standard. Then we transform this description into the chosen *formalization* notation and explicitly identify which *storage variable*s are evaluated by this description. Finally, we compare our formalized description with the baseline implementation to validate that it correctly expresses the expected property. All results were categorized into the outcome categories presented in Section 3.6: *Verified*, *Falsified* and *Inconclusive*

5.1.1 Verification Using Certora

The *Certora Prover* examines smart contracts with respect to their specifications in *Certora*'s *Verification Language CVL* by checking for violation of properties expressed in each of the two major constructs supported by CVL: Invariant Blocks which represent a set of state properties that must be true at all times; *Rules* representing precondition/postcondition

contracts on specific function calls. While Foundry verifies correctness using a random ordering of method calls, the Certora verifier conducts *deductive proof* attempting to demonstrate that no legal *execution path* of the contract can violate a specified property as opposed to merely demonstrating whether there are violations [25]. As such, Certora is the most commonly used *theorem prover* for *Ethereum* in this thesis, per Section 3.3.1.

The complete test file is available in the accompanying GitHub repository[3]. The following steps reproduce the verification of environment.

Install Python and Certora CLI

```
sudo apt install pipx -y
pipx install certora-cli
pipx ensurepath
```

Install Java

```
sudo apt install default-jdk -y
```

Install and configure Solidity compiler

```
pipx install solc-select
solc-select install 0.8.20
solc-select use 0.8.20
```

Set Certora API key

```
export CERTORAKEY=your_api_key_here
```

Run verification

```
certoraRun src/TestToken.sol \
  --verify TestToken:ERC20Invariants.spec \
  --solc solc
```

Target and Setup

The verification targets were *OpenZeppelin*'s *ERC20.sol* v5.2.0 committed in ACD4FF74. Since the *ERC20.sol* the "base" has declared *_mint* as *internal* with no intrinsic access control

[3] The ERC-20 Certora verification artefacts are available at: https://github.com/mabubakr1113/token-standards-formal-verification/tree/main/ERC-CERTORA-VERIFICATION

for itself, a specific *TestToken.sol* contract that directly inherits from *ERC20.sol* will be the actual verification target. This contract includes explicit minter authorizations prior to the execution of the *public mint* or *burn* functions, which makes *ERC20-I6* somewhat verifiable through this intermediate layer. The *Certora Prover* was called using the Solidity compiler v.0.8.20 by the Certora CLI v.8.13.

Specification

Nine rules were specified in CVL targeting the six ERC-20 invariants defined in Section 4.2.3. Following the specification procedure of Section 3.5, each rule was first described in plain language, then encoded in CVL, then reviewed against the OpenZeppelin source to confirm correctness. The methods block declares all relevant functions and marks totalSupply, balanceOf, and allowance as envfree since they do not depend on the transaction environment. The nine rules and their target invariants are described below.

1. transferPreservesSupply targets ERC20-I3. It records totalSupply before calling transfer and asserts it is unchanged after. No preconditions beyond the symbolic environment are required because the rule applies to all possible inputs.

2. transferFromPreservesSupply is the ERC20-I3 variant for transferFrom. It follows the same structure but calls transferFrom with three symbolic parameters.

3. transferFromRespectsAllowance targets ERC20-I4. It requires that the allowance before the call is less than max_uint256 to exclude the infinite approval case, requires the allowance is sufficient for the transfer, and asserts the allowance decreases by exactly the transferred amount after the call.

4. infiniteApprovalUnchanged targets the infinite approval exception of ERC20-I4. It requires the allowance equals max_uint256 before the call and asserts it remains unchanged after transferFrom. This rule directly encodes the OpenZeppelin implementation-defined behavior discussed in Section ERC20-I4.

5. onlyMintBurnChangeSupply targets ERC20-I5 using a universal method quantifier over all public functions. It asserts that if totalSupply changes after any function call, that function must be either mint or burn.

6. transferBalanceConsistency targets ERC20-I1 for the transfer case. It records sender and recipient balances and totalSupply before the call and asserts all three updates correctly after transfer.

7. mintBalanceConsistency targets ERC20-I1 for the mint case. It asserts that after mint, the recipient balance and totalSupply both increase by exactly the minted amount.

8. burnBalanceConsistency targets ERC20-I1 for the burn case. It asserts that after burn, the caller balance and totalSupply both decrease by exactly the burned amount.

Results

Table 11 presents the Certora Prover results for the ERC-20 invariants using the outcome categories defined in Section 3.6.

Rule	Invariant	Result	Time
transferPreservesSupply	ERC20-I3	Verified	14s
transferFromPreservesSupply	ERC20-I3	Verified	8s
transferFromRespectsAllowance	ERC20-I4	Verified	13s
infiniteApprovalUnchanged	ERC20-I4	Verified	9s
onlyMintBurnChangeSupply	ERC20-I5	Inconclusive	8s
transferBalanceConsistency	ERC20-I1	Inconclusive	3s
mintBalanceConsistency	ERC20-I1	Inconclusive	3s
burnBalanceConsistency	ERC20-I1	Inconclusive	3s
ERC20-I6 mint authorization	ERC20-I6	Inconclusive	-

Table 11: Certora Prover verification results for ERC-20 invariants using outcome categories from Section 3.6

Analysis of Results

Four rules were confirmed Verified and five rules were Inconclusive. All four rules verified, transferPreservesSupply, transferFromPreservesSupply, transferFromRespectsAllowance, and infiniteApprovalUnchanged, are reasoning about one storage variable at a time. transferPreservesSupply and transferFromPreservesSupply verify ERC20-I3 by showing there

is no valid execution path from either transfer or transferFrom to reach the _totalSupply storage slot. transferFromRespectsAllowance verifies the finite approval case of ERC20-I4 by showing the allowance reduction postcondition can be proved for all inputs that satisfy both the preconditions. infiniteApprovalUnchanged verifies that the OpenZeppelin specific behavior for infinite approvals is properly preserved.

All five of the Inconclusive results corresponded to exactly two different issues. The first was the onlyMintBurnChangeSupply rule. Only this one has been identified by Certora's universal method quantifier as having string-returning methods, i.e., name() and symbol(), for which it cannot currently provide a correct proof in this case. This is a known issue with the current version of the tool and represents an example of the tool limitation threat identified in Section 3.8. The second were the three balance consistency rules: transferBalanceConsistency, mintBalanceConsistency, and burnBalanceConsistency. These have failed because Certora can reason over all possible uint256 values and identify an arithmetic overflow as a viable path without explicitly including an overflow guard assumption. As previously mentioned, these are not bugs within the OpenZeppelin implementation; they represent valid conclusions regarding the specification model. Therefore, it is necessary to include explicit overflow assumptions if you intend to reason about combined balance and supply updates using CVL. For instance, adding require balanceOf(to) + amount <= max_uint256 as a precondition will remove the failures listed above; however, this will add an additional assumption that must be independently validated, i.e., related to the soundness of assumptions dimension in Section 3.7.

ERC20-I6 does not conclude because the base ERC20.sol contract has declared _mint as internal with no access control as described in ERC20-I6 section. The invariant can only be proven at a deployment contract level, which is where the authorizing logic is implemented [25]. Certora's results show that deductive verification produces better assurances of preservation for both supply and allowance properties than invariant fuzzing, however the deductive process requires considerable care to be exercised on the use of arithmetic assumptions and function type constraints, especially when employing universal quantification. The results obtained are in line with those found by Bhargavan et al., regarding the difference between specification at a source code level versus having full verification coverage [7].

5.1.2 Verification Using Foundry

Foundry is an open-source development environment for Solidity smart contracts and a testing platform designed to support *invariant fuzzing* using *Forge*'s forge test command [26]. The testing approach used by *Foundry* differs from that used by *Certora*, as instead of attempting to produce *deductive proofs* on all possible paths to prove that the contract is correct, Foundry uses the concept of random sequences of function calls. After each function call Foundry checks whether the declared *invariant* functions have been called correctly. Thus, *Foundry* can be viewed as a complementary tool to Certora; while Certora provides *deductive proof* of property correctness, *Foundry* generates counterexamples through the use of a brute force search over many different sequences of executions [26].

The complete test file is available in the accompanying GitHub repository[4]. The following steps reproduce the verification of environment.

Install Foundry

```
curl -L https://foundry.paradigm.xyz | bash
source ~/.bashrc
foundryup
```

```
mkdir erc20-verification && cd erc20-verification
forge init
forge install OpenZeppelin/openzeppelin-contracts@v5.2.0
```

Create project and install OpenZeppelin v5.2.0

Configure remappings in foundry.toml

Run invariant tests

[4] The ERC-20 Foundry verification artefacts are available at: https://github.com/mabubakr1113/token-standards-formal-verification/tree/main/ERC-FOUNDRY-VERIFICATION

The full campaign completes in approximately 25 seconds on a standard development machine

```
[profile.default]
src = "src"
out = "out"
libs = ["lib"]
remappings = [
    "@openzeppelin/contracts/=lib/openzeppelin-contracts/contracts/"
]
```

```
forge test -vv
```

Handler-Based Invariant Testing

In making key decisions related to design within the above referenced specification, it has been determined that the specification will utilize a *handler contract*. By doing so, rather than allow the *fuzzer* for *Foundry* to directly call functions with arbitrary input on the token contract, the *handler* limits the *fuzzer* to a specific set of four actor addresses and utilizes *Foundry*'s bound() utility to limit the amount to be transferred. This limits the *fuzzer* from utilizing "runs" to perform trivially reverted calls such as transfer funds from an account that holds no balance and allows for the campaign to focus on meaningful *state transition*. In addition, the *handler* maintains two running counters totalMinted and totalBurned, which are utilized directly in the *ERC20-I3* and *ERC20-I5* assertion statements.

Without the *handler*, the naive specification gave me two *false positives*. The *fuzzer* correctly determined that calling mint on an untracked address changed the sum of three fixed addresses, which would diverge from totalSupply. This was not a bug in OpenZeppelin; it was a specification error: *invariants* should be defined over all addresses receiving tokens, not simply some predefined subset. As stated above, this finding demonstrates consistency with the specification quality threats discussed in Section 3.8; a formally incorrect *invariant* will produce findings based upon errors in the specification as opposed to errors in the implementation.

Specifications

Five *invariant* functions were specified according to the process for specifying *invariants* in Section 3.5. Each was initially explained in plain terms and subsequently written into a *Solidity* function starting with *invariant_*. Then each was compared with the equivalent in the *OpenZeppelin* library:

1. *invariant_I1_supplyEqualsSumOfBalances* specifies that totalSupply is equal to the sum of the four actor balances at all times. This is an example of the bounded inductive form of ERC20-I1 as presented in Section 4.2.3.

2. *invariant_I2_nonNegativeBalance* ensures that no actor's balance goes below zero; this corresponds to ERC20-I2.

3. *invariant_I3_supplyConservation* states that totalSupply will be equal to the initial supply plus *handler.totalMinted* minus *handler.totalBurned*; this defines ERC20-I3 and ERC20-I5 simultaneously by definition.

4. *invariant_I4_allowanceBound* represents ERC20-I4 structurally via the handler design.

5. *invariant_I5_onlyMintBurnChangeSupply* employs the same assertion as *invariant_I3_supplyConservation*; it confirms that no function other than *mint* or *burn* alter *totalSupply*.

Results

All five invariants passed with no counterexample found. Table 12 presents the confirmed results using the outcome categories defined in Section 3.6.

Invariant	Result	Runs	Calls	Reverts
ERC20-I1: Supply equals sum of balances	Verified	256	128,000	0
ERC20-I2: No negative balance	Verified	256	128,000	0
ERC20-I3: Supply conservation	Verified	256	128,000	0
ERC20-I4: Allowance bound	Verified	256	128,000	0
ERC20-I5: Only mint and burn change supply	Verified	256	128,000	0

Table 12: Foundry invariant fuzzing results for ERC-20, 256 runs per invariant, 128,000 total calls each

Analysis of Results

All five *invariants* were tested over 128,000 calls to their respective *handlers*. The handler design resulted in zero reverts across those 128,000 calls. There are no reverts as the *fuzzer* had exhausted all of its 128,000 calls on legitimate *state changes* and not meaningless invalid states. Each of the five handler functions was called approximately 25,500 times during each of the five *invariant* campaigns; this has provided equal distribution across approve, burn, mint, transfer and transferFrom. The total test campaign time was 25.61 seconds. Additionally, the lack of reverts provides supporting indirect evidence of ERC20-I4. As long as the *handler* does not make a delegated transfer from when it exceeds the allowed *allowance* or balance, due to the fact there were no reverts, the OpenZeppelin transferFrom function will properly handle all valid delegated transfers. A *postcondition* check for allowable reductions to *allowances* prior to each call can be done using an assertion on *allowance* before and after the call; however, since *Foundry* is based upon an *invariant testing* paradigm and therefore cannot perform such assertions directly as demonstrated in Section 5.1.1, they are better suited for certification by Certora.

Like other *fuzz-based* techniques, a pass does NOT imply that we have formally shown a property violation. *Foundry* is unable to generate all possible *execution paths* and may fail to identify violations which occur solely under unique input configurations not generated during the *Foundry* campaign [7]. This corresponds to the *coverage* metric from Section 3.7. In this section, I rate *Foundry* as *Partially Supported*; it tests a portion of the *state space* rather than all portions of the *state space* represented by the *verification model*. A complete *deductive* method to test these *invariants* can be obtained using *Certora* as described in Section 5.1.1; thus, when used collectively they are able to present stronger collective evidence than each tool would separately.

5.1.3 Verification Using KEVM

KEVM is an executable *formal semantics* for the Ethereum Virtual Machine developed with the *K Framework*. *KEVM* differs from *Certora* in that it does not verify *Solidity* source code but instead verifies reachable states of EVM bytecode. In addition to this difference, *KEVM* is the only one of these three tools to perform actual *reachability analysis* on EVM bytecode. Therefore, *KEVM* provides the strongest deployment level assurances of all three. *KEVM* in Section 3.3.1 because *KEVM* bridges the gap between *source-level* proofs and *bytecode-level*

guarantees, a limit Bhargavan et al. have pointed out as being a key issue with verifying only the source code [7].

Specifications

Using the formalism from Section 3.5, I have encoded the *KEVM* specification as a *reachability claim* in *K* notation to target *ERC20-I3*: "*transfer* does not change *totalSupply*." I modeled the *EVM* storage layout, i.e., *slots*, of the *OpenZeppelin ERC20.sol* contract directly into the model and used the deterministic *storage slot* assignment rules provided by *Solidity*. Thus, slot 0 contains _balances with keys hashed using keccak256, and slot 1 contains _totalSupply. My *reachability claim* has three parts. The *pre-state* has symbolic variables assigned for the sender's balance BAL_FROM, the recipient's balance BAL_TO, the transfer amount AMOUNT, and the current total supply SUPPLY. In this part, the *post-state* asserts that slot 0 containing _balances[caller] decrease by AMOUNT, slot 0 containing _balances[to] increases by AMOUNT, and slot 1 containing _totalSupply does not change. Finally, the *requires clause* lists all necessary *preconditions* which include the sender and recipient being different addresses, the sender having at least a balance of BAL_FROM, AMOUNT > 0, and the recipient's new balance will not exceed an *overflow* after the transfer.

The *postcondition* on *storage slot* 1 which states SUPPLY => SUPPLY indicates that prior to and subsequent to execution the *total supply* at the _totalSupply slot will be unchanged. Any such slot write would be rejected by *KEVM*; thereby establishing an evidence-based *bytecode level* proof for *ERC20-I3*.

Three successive efforts were attempted to create a functional KEVM execution environment on the WSL2 Ubuntu 24.04 development machine used throughout this study. Each effort is described below.

Attempt 1: Direct nix installation

KEVM v1.0.678 was successfully installed via the kup package manager using the K framework install script:

```
bash <(curl https://kframework.org/install)
. /nix/var/nix/profiles/default/etc/profile.d/nix-daemon.sh
kup install kevm
```

Installation completed successfully and kevm version confirmed KEVM v1.0.678. However, when attempting to kompile the specification:
The following error was produced:
Exit Code 113 is indicative of a failed link. *KEVM*'s *nix*-based build utilizes bundled versions

```
subprocess.CalledProcessError: Command kompile returned
non-zero exit status 113
```

of *OpenSSL* and *secp256k1* from the *nix* store as part of krypto.a. The *nix* store paths are available for access in *WSL2*; however, the link creation process between *nix*-compiled crypto libraries and the *LLVM* backend fails due to an incompatible version of *LLVM* that is utilized by both *WSL2*'s kernel and *nix*.

Attempt 2: System C++ library installation

To resolve the linking failure, the required C++ and cryptographic libraries were installed at the system level.

```
sudo apt install -y libssl-dev libsecp256k1-dev \
  libc++-dev libc++abi-dev build-essential cmake
```

The kompile command was attempted again to verify whether the prior error was a one-time event. An identical exit code of 113 was received, confirming that *KEVM*'s *nix* build system does not utilize system-wide libraries; instead, it bundles its own libraries using *nix store*. Therefore, installing system libraries would have no impact on the install process.

Attempt 3: Docker container

A Docker-based approach was attempted as the final alternative. The official KEVM Docker image was searched on Docker Hub,

```
sudo docker pull runtimeverificationinc/kevm:latest
sudo docker pull runtimeverification/kevm:latest
```

A "not found" or "access denied" message was generated by both commands. The *DockerHub* website for *RuntimeVerificationInc/KEVM* confirms the container image has not been updated in greater than two years and therefore, is no longer supported nor available for public use.
It was next determined that the *Kontrol* Docker image, *Runtime Verification*'s integrated *KEVM* and *Foundry* tool, is an active solution.
Exit code 137 signifies that the process has been killed by the operating system because it does not have enough memory. The compilation using the *KEVM LLVM* backend is an

extremely memory-intensive process requiring a high amount of RAM for completion. As such the *Docker* container running in *WSL2* did not have enough memory assigned to the *WSL2* virtual machine to finish the compilation. This corresponds to the default behavior of memory allocation in *WSL2* to cap available RAM to a fraction of the host system's memory if not specified differently in a .wslconfig file.

Results

In each of the three approaches taken to test the *KEVM* verifier with *ERC20-I3*, it resulted in an *Inconclusive* verification result due to environmental limitations established in Section 3.6. Each of the three different failure types that occurred were related to environments rather than specifications; i.e., a *nix LLVM* linking error during the direct installation, inability to resolve system libraries properly during the *C++* approach, and an out-of-memory exit condition during the *Kontrol Docker* based approach. The *K* specification has been completely developed and syntactically correct; the *K* specification accurately captures both the *EVM* storage structure and the *postcondition* about *supply conservation*. No counterexample was generated nor did any proof occur since none of the three approaches completed the *verification* execution step.

This conclusion has significant implications on the assessment of *KEVM*'s usefulness as shown in previous section. *KEVM* is rated as *Not Supported* to verify, since it does not have any results, *Fully Supported* to be expressive, because the *K* specification successfully and completely expresses the *invariant*, and *Not Supported* concerning specification costs when deployed in an environment such as this one, due to the compilation barrier preventing execution of the specifications. In addition, my experience confirms the findings by Hildenbrandt et al., who state that although *KEVM* offers the best theoretical guarantee among the three *Ethereum* tools available, its application can only succeed in a native *Linux* environment with enough memory, and a compatible version of the *LLVM* compiler [24].

5.2 Rust-Based Verification

5.2.1 Verification Using Kani

Kani is a *bounded model checker* that can be used to check *Rust* for correctness using *symbolic input* proofs and *proof harnesses*. Like *fuzzing* tools, *Kani* tests every path through code within a given bound, proving that some property will always hold or providing a counterexample, but unlike *fuzz testing* it does so by systematically checking all potential

execution paths, rather than randomly selecting them. *Kani* was chosen for this thesis based on how closely the *harness*-based methodology employed by *Kani* would mirror the *SPL Token* instruction model. In the *SPL Token* instruction model, each operation executes with explicit account information being provided. Therefore, like the *bounded model checking* method employed by *Kani*, the methodology used here to evaluate instruction logic at the function level fits well into the reduced scope strategy outlined in Section 3.2.2.
Kani version 0.67.0 was installed on WSL2 Ubuntu 24.04 using the following steps.

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
cargo install --locked kani-verifier
cargo kani setup
```

A separate, self-contained *verification crate* was set up under ~/spl-kani-verification to use version 1.95.0 of the stable *Rust* compiler. The *verification crate* did not depend upon any other crates. Rather than import the entire *SPL Token* crate into the *verification crate* as an alternative, the basic structure of the *SPL Token* accounts were modeled directly in the *verification crate*'s code. Using a direct modeling approach for these structures allowed for avoiding potential problems related to compatibility with the tool chain, and it also helped keep the focus of the verification to the most important part: how the *SPL Token* instructions behave. The complete test file is available in the accompanying GitHub repository[5].

Account Model

The two *structs* represent the two kinds of *SPL Token* accounts that matter. MintAccount represents supply, is_initialized, and mint_authority. TokenAccount represents amount, is_frozen, is_initialized, mint_key, owner, delegate, has_delegate, and delegated_amount. Those are direct representations of each of the accounts shown in Section 4.3.1 and Figure 5. Public keys are used here as symbolic u64 values so we can reason about both equality and authorization without needing a complete *crypto* representation.

Specifications

In total, seven *proof harnesses* for proof of the six *SPL Token* invariants were written using the specifications found in Section 4.3.3. Two separate *proof harnesses* were required for

[5] The SPL Token Kani verification artefacts are available at: https://github.com/mabubakr1113/token-standards-formal-verification/tree/main/SPL-KANI-VERIFICATION

SPL-I1, due to the fact that the MintTo and Burn operations both impact the token supply but in different ways. A single *proof harness* will be used for each operation since they require their own induction arguments. All three *proof harnesses* use kani::any() to allow *Kani* to provide *symbolic inputs*, i.e., to allow *Kani* to explore all valid account states with respect to an upper bound on the model, and kani::assume() was also used in the *SPL-I1* proof harnesses to limit the range of input values less than u64::MAX / 2. These limitations are intended to prevent arithmetic overflows from being explored by *Kani*; therefore, this is considered a *soundness of assumptions* limitation as discussed in Section 3.7.

The *proof harnesses* and their respective *invariants* are listed below.
verify_spl_i3_transfer_supply_conservation is to verify the *SPL-I3* invariant by checking if there has been no change in mint.supply after a successful *transfer*.
verify_spl_i2_no_underflow is to verify the *SPL-I2* invariant by checking that the source of tokens will never have an increased value after either a successful or unsuccessful *transfer* execution. verify_spl_i1_mint_balance_consistency and verify_spl_i1_burn_balance_consistency test the MintTo and Burn cases for *SPL-I1* respectively by verifying that both mint.supply for MintTo and destination.amount for MintTo, or source.amount for Burn do each increase or decrease by the same amount, the minted or burned amount. verify_spl_i4_mint_authority_enforcement verifies the *SPL-I4* invariant by testing whether an invalid *mint authority* generates an error message and ensures that mint.supply has not changed. verify_spl_i5_token_account_mint_ownership tests the *SPL-I5* invariant by first assuming that the mint_key from the source and destination accounts is different; then it asserts that the *transfer* operation will result in an error being generated. verify_spl_i6_delegate_amount_enforcement assumes that the transfer amount is greater than the amount that was delegated; then it asserts that this will generate an error.

Results

All seven harnesses were verified successfully with zero failures across 597 total checks. The complete results are shown in Table 13.

Harness	Invariant	Checks	Failed	Result	Time
verify_spl_i3_transfer_supply_conservation	SPL-I3	85	0	Verified	0.068s

verify_spl_i2_no_underflow	SPL-I2	84	0	Verified	0.067s
verify_spl_i1_mint_balance_consistency	SPL-I1	77	0	Verified	0.249s
verify_spl_i1_burn_balance_consistency	SPL-I1	59	0	Verified	0.309s
verify_spl_i4_mint_authority_enforcement	SPL-I4	70	0	Verified	0.091s
verify_spl_i5_token_account_mint_ownership	SPL-I5	90	0	Verified	0.069s
verify_spl_i6_delegate_amount_enforcement	SPL-I6	132	0	Verified	0.189s
Total	**SPL-I1 to SPL-I6**	**597**	**0**	**Verified**	**~1.07s**

Table 13: Kani verification results for SPL Token invariants

Analysis of Results

The six *SPL Token* invariants were confirmed to be valid through 597 tests using *SPL Token* with no errors occurring over an approximate time of 1.07 seconds. A two-harness system was used to confirm *SPL-I1* as the inductive proof of MintTo and Burn had to be created independently; one to show that supply is increasing by an equal measure of the destination value, and the other showing that supply is decreasing by an equal measure of the source value. The necessity for this type of separation reflects the structural challenges associated with validating *cross-account balance consistency* which are described in Section 4.3.3 and can occur when an invariant must validate data from more than one *struct* rather than a single *contract mapping*.

The *SPL-I6* harness produced the largest number of checks, 132, due to the large number of *branching conditions* which were present within the *delegate authorization* path. The *SPL-I1* Burn harness was the slowest at 0.309 seconds, as *Kani* checked all possible combinations

that satisfy the assumption constraint initial_supply >= burn_amount. A *soundness* limitation exists with the kani::assume() used to restrict arithmetic inputs in the *SPL-I1* harnesses; therefore, the verification results are true assuming that supply and amount values do not exceed u64::MAX / 2. While this is a plausible real-world token deployment, it also means the harness does not cover the entire range of u64. This is reflected in the *soundness of assumptions* rating in Section 3.7.

The *verification model* eliminates the need for several of the *Solana* runtime's features such as *gas mechanics*, *account rent*, *cross-program invocation*, and *parallel execution* scheduling. Since these are abstractions, the harnesses will verify the *arithmetic logic* and *authorization logic* of the instruction models, but they cannot be used to verify behavior that would have to occur in a full *Solana* runtime environment. This is consistent with the *harness modeling threat* identified in Section 3.8.

5.2.2 Verification Using Prusti

Prusti is a *deductive verifier* for *Rust* based upon the *Viper* verification infrastructure. The tool enables users to add *precondition* annotations to *Rust* functions with #[requires(...)], and *postcondition* annotations to *Rust* functions with #[ensures(...)]. Following this, *Prusti* will prove statically that the implementation of the function satisfies the *postcondition(s)*, for every input satisfying the *precondition(s)*. *Prusti* was chosen as a result of the natural correspondence of the specification style employed by *Prusti*, contracts based on pre/post conditions, to the *invariant* structure of *SPL Tokens* expressed through *pre/post condition* invariants within Section 4.3.3. Additionally, while both tools are capable of expressing *pre/post condition* invariants, *Prusti* offers a greater degree of expressive capability relative to *Kani* in order to express complex invariants of account relationships. The complete test file is available in the accompanying GitHub repository[6].

Three installation attempts were made before a working configuration was achieved.

1. Attempt 1 cargo install: The cargo install command for cargo-prusti was unable to locate cargo-prusti in the registry crates.io. Because it is not available on crates.io,

[6] The SPL Token Prusti verification artefacts are available at: https://github.com/mabubakr1113/token-standards-formal-verification/tree/main/SPL-PRUSTI-VERIFICATION

Prusti will need to be installed by either downloading one of the precompiled binary releases of *Prusti* or compiling from source.

2. Attempt 2 dependency resolution via crates.io: After download of the precompiled binary release v-2024-02-28-1817 and attempt to include prusti-contracts = "0.1.0" as a dependency, cargo failed due to: failed to parse the edition key; this version of cargo is older than the 2024 edition. Only editions prior to 2021 are supported by the nightly-2023-09-15 toolchain that comes bundled with the cargo installation. Since its edition support has been raised to 2024, it would be impossible to resolve dependencies using this toolchain.

3. Attempt 3 *git* dependency: Using *prusti-contracts* as part of the *Prusti git* repo tag *v-2024-02-28-1817* created the same Edition 2024 error when it was resolved by a transitive dependency. Solution: *Prusti* was run on this project directly with *prusti-rustc* with no external crate dependence; I then used an *extern crate* for *prusti_contracts* to use the contract library that came with *Prusti*. Additionally, after running the command to make *z3* executable at *~/prusti-release/viper_tools/z3/bin/z3* using *chmod +x*, I was able to run *Prusti*.

Target and Setup

To verify the system, *I* targeted an independent crate *spl-prusti-verification* that utilized the *Rust* tool chain file *rust-toolchain* included in the *Prusti* release and pinned the project to *nightly-2023-09-15*. *I* used *Prusti* version 0.2.2 at commit *2d14265*, which was built at 2024-02-28. In addition to the *SPL Token* account structure described in Section 4.3.1, verification was performed as follows:

```
~/prusti-release/prusti-rustc --edition=2021 --
crate-type=lib src/lib.rs
```

Specifications

Eleven functions were defined using *Prusti*'s contract macros #[requires] and #[ensures]. The specification set consists of a combination of correct function implementations and broken function implementations, to provide an example for demonstrating how *Prusti* can identify real *invariant* violations as well as validate correctly implemented code. The correct implementations were created targeting *SPL-I3* by utilizing *spl_transfer*, *SPL-I4* by utilizing *spl_mint_to* and *spl_mint_to_unauthorized*, *SPL-I6* by utilizing *spl_delegated_transfer*, *SPL-I2* by utilizing

spl_debit_source, *SPL-I5* by utilizing *spl_validate_mint_match*, and *SPL-I1* by utilizing *spl_i1_mint_consistency* and *spl_i1_burn_consistency*.

The wrong implementations contain errors that are designed to evaluate *Prusti*'s ability to detect them. *spl_transfer_buggy* breaks *SPL-I3* as it subtracts from mint.supply while doing a transfer. *spl_mint_to_buggy* breaks *SPL-I4* in that it mints but does not check the caller is authorized for the mint. *spl_delegated_transfer_buggy* breaks *SPL-I6* because it omits the amount of delegated_amount to be reduced after a delegated transfer has been made. *spl_i1_burn_buggy* breaks *SPL-I1* as it reduces only source.amount and does not reduce mint.supply.

Results

Prusti verified 13 items and detected 4 postcondition violations. The complete results are shown in Table 14.

Function	Invariant	Result	Violation
spl_transfer	SPL-I3	Verified	—
spl_transfer_buggy	SPL-I3	Falsified	mint.supply postcondition violated
spl_mint_to	SPL-I4	Verified	—
spl_mint_to_buggy	SPL-I4	Falsified	caller == mint_authority postcondition violated
spl_mint_to_unauthorized	SPL-I4	Verified	—
spl_delegated_transfer	SPL-I6	Verified	—
spl_delegated_transfer_buggy	SPL-I6	Falsified	delegated_amount postcondition violated
spl_debit_source	SPL-I2	Verified	—
spl_validate_mint_match	SPL-I5	Verified	—
spl_i1_mint_consistency	SPL-I1	Verified	—

spl_i1_burn_consistency	SPL-I1	Verified	—
spl_i1_burn_buggy	SPL-I1	Falsified	mint.supply and source.amount postconditions violated

Table 14: Prusti verification results for SPL Token invariants

Analysis of Results

Prusti confirmed the correctness of each correct implementation and correctly identified each of the four *invariants* that were violated. The four *falsifications* have a direct correspondence with real-world *SPL Token* vulnerabilities. The *spl_transfer_buggy* violation shows how *Prusti* detects unauthorized modifications to supply when transferring tokens, a vulnerability discussed in detail in Section 2.6.2. The *spl_mint_to_buggy* violation identifies how *Prusti* detects missing *authority* checks before minting tokens, as shown in *SPL-I4*. The *spl_delegated_transfer_buggy* violation illustrates how *Prusti* detects missing *delegated amount* reductions, as shown by *SPL-I6*. The *spl_i1_burn_buggy* violation also shows how *Prusti* detects missing logic from a *burn* action that does not reduce both the supply of the token and the *token account* balance, as specified by *SPL-I1*.

Kani vs. *Prusti*: compared to *Kani*, *Prusti*'s contract-based approach requires more annotation effort per function but produces more precise error messages that identify the exact *postcondition* violated and the line number of the originating function. *Prusti* is therefore better suited for auditing specific functions against known *invariants*, while *Kani* is better suited for exhaustive *bounded* exploration of all *execution paths*.

5.2.3 Verification Using MIRAI

MIRAI is an *abstract interpreter* for *Rust*'s *Mid-level Intermediate Representation* (*MIR*) and is used to analyze all potential paths of a program using *abstract interpretation*, unlike *Kani* that does *bounded model checking* and *Prusti* that performs *deductive verification*. *MIRAI* can be used to detect all reachable panics, as well as violation of contracts expressed with precondition! and postcondition! macro calls. In addition, *MIRAI* is able to identify failures at runtime including but not limited to *arithmetic overflows*, *out-of-bounds array access*, *unreachable code* and so on.

MIRAI version 1.1.9 was built from source from the facebookexperimental/MIRAI GitHub repository using the nightly-2025-11-21 Rust toolchain. The following dependencies were required.

Building MIRAI from source requires compiling Z3 from source as a dependency, which is a large C++ codebase. The build process took approximately 30 minutes on the WSL2 Ubuntu 24.04 environment used in this thesis.

```
apt install cmake clang -y
cd ~ && git clone https://github.com/facebookexperimental/MIRAI.git
cd MIRAI && rustup override set nightly-2025-11-21
nohup cargo +nightly-2025-11-21 install --locked --path ./checker >
mirai-build.log 2>&1 &
```

Target and Setup

A custom crate named *spl-mirai-verification* was generated in *Rust* version 1.95.0. When you invoke *MIRAI* as a *cargo* wrapper, through the command line argument RUSTC_WRAPPER=mirai cargo build, it will replace your standard *Rust* compiler with *MIRAI*'s *abstract interpreter* at compile time. The *mirai-annotations* crate includes the macros precondition! and postcondition! to be used for contract specifications. he complete test file is available in the accompanying GitHub repository[7].

Specifications

MIRAI's specifications are written differently than those used with *Kani* and *Prusti*. Instead of using *proof harnesses* or function level annotations as used by *Kani* and *Prusti*, *MIRAI*'s specifications use precondition! and postcondition! macros directly within the body of the functions they validate. In addition, when no specifications are given for code to be analyzed, *MIRAI* will perform an unannotated analysis of that code and find all reachable panics and all unreachable arithmetic issues. This feature is particularly useful as a first pass at analyzing the *SPL Token* instruction logic prior to writing explicit contracts. The verification specification includes the same six *SPL Token* invariants described in Section 4.3.3. Each invariant has two

[7] The SPL Token MIRAI verification artefacts are available at: https://github.com/mabubakr1113/token-standards-formal-verification/tree/main/SPL-MIRAI-VERIFICATION

parts. The precondition! macro will guard the entry point into a function while the postcondition! macro will assert the desired condition of the system after that function executes. *MIRAI*'s *verify* diagnostic can be activated via the MIRAI_FLAGS="--diag=verify" environment variable. This allows *MIRAI* to output all possible contract violations as opposed to only those that are certain.

Results

The *MIRAI* verification crate *spl-mirai-verification* re-models minting *SPL Token* accounts directly as described in Section 4.3.1 using public keys in u64 format. There are six correctly implemented functions that correspond to instructions in the model, four incorrect versions of these instruction functions which have been determined to be faulty as they use the same set of faults used in the *Prusti* experiment in Section 5.2.2, and an unannotated overflow-demonstrating function. *cargo mirai* performs a top-level, path-sensitive whole program analysis for all entry points in the crate. *cargo mirai* provided a list of all twelve functions in the crate using the *--print-function-names* flag including auto derived trait methods; *cargo mirai* also provided statistics with the *--statistics* flag indicating that the crate was analyzed and five diagnostics were emitted. Table 15 presents the MIRAI results for the SPL Token invariants and seeded faulty functions.

Function	Invariant	Source line	MIRAI result
spl_transfer	SPL-I3	—	Verified
spl_debit_source	SPL-I2	—	Verified
spl_mint_to	SPL-I1 (mint), SPL-I4	—	Verified
spl_burn	SPL-I1 (burn), SPL-I5	—	Verified
spl_validate_mint_match	SPL-I5	—	Verified

spl_delegated_transfer	SPL-I6	—	Verified
spl_transfer_buggy	SPL-I3	238	Falsified (mint.supply postcondition unprovable)
spl_mint_to_buggy	SPL-I4	256	Falsified (mint_authority postcondition unprovable)
spl_delegated_transfer_buggy	SPL-I6	278	Falsified (delegated_amount postcondition unprovable)
spl_i1_burn_buggy	SPL-I1 (burn)	296	Falsified (mint.supply postcondition unprovable)
spl_naive_credit	runtime safety	—	Reachable add-with-overflow panic

Table 15: MIRAI verification results for SPL Token invariants

Analysis of Results

MIRAI identified all six correct implementations of *SPL Token* functions and identified all four incorrect implementations as such. Since the fault set for this experiment was the same as those tested in the *Prusti* experiment in Section 5.2.2, the results from both experiments may be compared directly. Each tool identifies the same four violations to *invariants*. However, they reach these conclusions with different logic. *Prusti* proves each function against its contract through *deductive proof* using the *Viper* system. In contrast, *MIRAI* uses *abstract interpretation* of the *mid-level intermediate representations*. *MIRAI* then tracks abstract values down every possible path taken within a program until it finds where the *postconditions* are no longer proven to be valid. For example, *MIRAI* found that *spl_transfer_buggy* violates *SPL-I3* since *MIRAI* detected that there is an unauthorized write to mint.supply when transferring tokens. Similarly, *MIRAI* found that *spl_mint_to_buggy* violates *SPL-I4* since *MIRAI* detected that mint does not include *authority* checking. Further still, *MIRAI* found that *spl_delegated_transfer_buggy* violates *SPL-I6* since *MIRAI* detected a delegation transfer which will fail to decrease delegated_amount. Finally, *MIRAI* found that *spl_i1_burn_buggy* violates *SPL-I1* since *MIRAI* detects that a *burn* has been performed and decreased source.amount without decreasing mint.supply. As stated above, these correspond to the specific *SPL Token* vulnerabilities described in Section 2.6.2.

The key characteristic of the results generated by the *MIRAI* tool is the fifth type of diagnosis. Without any annotation, *MIRAI* reports an *arithmetic overflow* to be reachable in function *spl_naive_credit* where the addition operation is unchecked. Neither *Kani* nor *Prusti* report such runtime failures such as *arithmetic overflows* without having either an explicit *harness* or contract. As mentioned above, this is the panic-detecting run without annotations discussed in Section 5.2.3 and represents the primary complementary feature of *MIRAI* relative to the other two tools.

One behavior of *MIRAI* is worth noting. When the same untreated addition sits in a public function that has an overflow that depends on caller-supplied inputs, by default *MIRAI* does not issue any warning for this behavior but rather promotes the non-overflow condition to an implicit *precondition* of the public interface. The overflow will only be observable when it is certain and reachable as in the demonstration driver, or when a stricter diagnostic level is requested which requires such *preconditions* to be made explicit. This is consistent with *MIRAI*'s conservative reporting policy under which it suppresses findings it judges as potential *false positives* and it also explains why the overflow is reported for the demonstration path and not for the *spl_naive_credit* arithmetic alone. Likewise, both *SPL-I1* contracts bind the token supply and amount to less than half of u64::MAX so that the additive *postcondition* arithmetic cannot overflow either. This is the same *soundness of assumptions* limitation recorded for *Kani* in Section 5.2.1; the guarantees hold for realistic token supplies and amounts within that bound, but they do not cover the full u64 range. Similarly, *abstract interpretation* models also abstract away from *Solana* runtime features identified in Section 3.8 including *account rent*, *cross-program invocations* and *parallel execution scheduler*, therefore the analysis reasons about instruction level *arithmetic* and *authorization logic* and not full runtime behavior.

Considering these three *Rust* tools collectively, *Kani* performs a type of *bounded model checking* and is exhaustive in terms of its input bounds; *Prusti* performs *deductive verification* to provide the most specific diagnostic information regarding which violation was made to which *postcondition* as well as the source line number; *MIRAI* performs an *abstract interpretation* of the entire function including matching *Prusti*'s ability to detect contract violations in addition to being able to identify reachable runtime panics without any specification. Overall, they complement each other by providing: *Kani* for performing exhaustive *bounded* explorations, *Prusti* for auditing contracts precisely and identifying

violations, and *MIRAI* for detecting reachability issues along paths leading to panic conditions.

5.3 Consolidated Verification Results

This section summarizes the results from the tool-by-tool analysis of Sections 5.1 and 5.2 using the five evaluation criteria identified in Section 3.7, namely *verifiability*, *expressiveness*, *specification effort*, *coverage*, and *soundness of assumptions*. All five criteria were rated using one of three support levels defined in Section 3.7: *Fully Supported*, *Partially Supported*, and *Not Supported*, which was found preferable to a numeric score given the qualitative nature of the judgments. The ratings were determined by the results contained in this chapter. Specifically, the per-*invariant Verified*, *Falsified*, and *Inconclusive* results determine the *verifiability* rating, and the use of reconstructed account models for the *Rust* tools, as outlined in Section 3.2.2, is used to justify both the *verifiability* and *coverage* ratings. Table 16 provides an overall assessment of all six tools. Only the ratings are presented here; a full justification for each rating along with cross-platform comparisons is provided in Chapter 6.

Tool	Verifiability	Expressiveness	Specification effort	Coverage	Soundness of assumptions
Certora	Partially supported	Fully supported	Partially supported	Fully supported	Partially supported
Foundry	Fully supported	Partially supported	Partially supported	Partially supported	Fully supported
KEVM	Not supported	Fully supported	Not supported	Not supported	Fully supported*
Kani	Partially supported	Fully supported	Partially supported	Partially supported	Partially supported
Prusti	Partially supported	Fully supported	Partially supported	Fully supported	Partially supported
MIRAI	Partially supported	Fully supported	Partially supported	Fully supported	Partially supported

Table 16: Consolidated tool effectiveness across the five evaluation criteria of Section 3.7

* KEVM soundness is fully supported in principle because it targets formal EVM semantics, but no completed result was obtained due to setup/resource limitations.

Two trends can be immediately identified from Table 16, both of which are discussed in further detail in Chapter 6. The first trend is that no single tool received a rating of *Fully Supported* across all dimensions, indicating that the dimensions reflect genuine trade-offs among the various tools rather than identifying a single best tool. The second trend is that the *Solidity* and *Rust* tools received different ratings for the *soundness of assumptions* dimension, due to the fact that the *Solidity* tools were applied against the actual *OpenZeppelin* source code whereas the *Rust* tools were applied against reconstructed account models. This difference in *soundness of assumptions* ratings is analyzed as the *target fidelity* threat in Section 6.2.

6 Discussion

6.1 Tool Effectiveness for Fungible Token Invariants

Tool performance on fungible token invariants cannot be solely assessed by determining if a property has been verified, falsified, or left inconclusive. An assessment needs to evaluate both how accurately the tool models its verification logic in relation to the structural elements of the invariant being verified. Results presented in Chapter 5 indicate that the degree of difficulty with which an analyst verifies a fungible token invariant differs depending upon whether the invariant presents a "single variable" preservation property, a "relational balance-and-supply" property, a "delegated authorization" property, or some combination of those three types. This distinction is relevant because not all types of reasoning-based formal verification tools have equal strength. Source-level deductive tools are most effective when verifying explicit functional specifications. Fuzzing tools are more effective at discovering practical counterexamples. Static analyzers such as Securify are best used to identify and flag known vulnerability patterns as opposed to formally proving functional invariants [39]. Bytecode-level tools provide the strongest assurances about deployment-level correctness but also require analysts to contend with additional complexity during operation [24], [40]. Therefore, tool effectiveness in this thesis should be understood as a relationship between invariant structure, specification style, and platform architecture rather than as a simple ranking of tools.

ERC-20 was the strongest tool using the source-level invariant reasoning approach on Solidity, because Certora's rule-based specification model matches the pre/post condition form used by transfer, transferFrom, and allowance properties [25], [37]. The rules proven for preservation of supply through transfer, preservation of supply through transferFrom, finite reduction of allowances, and infinite preservation of approvals demonstrate that the tool performs very well when an invariant relates to a clearly definable storage relation defined over fewer than a handful of contract variables. These results align well with the design rationale for tools such as solc-verify, which discharge functional correctness properties most effectively when they have been encoded into the contract abstraction [37], and with the general observation that the closer the analysis is performed to the contract abstraction, the less distance exists between the stated property and the code being verified [38]. However, Certora failing in balance inconsistency is significant because it proves stronger symbolic

reasoning and also shows weak specifications; as it did not only fail but exposed weak arithmetic overflow assumption was not present in the model [25], [37].

Therefore, the Certora results support a cautious interpretation of deductive verification. First, while Certora may provide greater assurance than fuzzing by proving there are no legal execution paths violating an ERC-20 property [25], [37], its inability to prove or disprove some cases demonstrates that a deductive verifier is only as reliable as the assumptions or modeling constraints provided to it [38]. Second, since ERC20-I1 must reason simultaneously about sender balance, receiver balance, and total supply to ensure balance consistency, even though OpenZeppelin's code does not necessarily represent an error, the specification must still rule out all mathematically feasible overflow paths if the intent is to use this proof to validate the intended token-level property [25]. Thus, the usefulness of Certora lies not only in which properties it verifies, but also in forcing the verifier to make all implicit assumptions explicit.

Foundry demonstrated another type of performance. While Certora provided evidence that no violations occurred, Foundry shows invariant preservation through testing against generated sequences of transactions [41], [42]. As discussed in Chapter 5, using the handler-based Foundry campaign the first five ERC-20 invariants were tested. All five invariants passed the test indicating that the chosen OpenZeppelin ERC-20 behaviors remain consistent across the generated state transitions. The benefit to the developer is that fuzzing can be added to normal development processes and will find errors in either the developer's code or specifications. Additionally, the successful use of Foundry's handlers was key to its ability to correctly identify issues with token implementations. This also fits within the smart contract fuzzing literature. Tools such as ContractFuzzer, Harvey, and sFuzz have shown that while fuzzing is very effective at discovering vulnerabilities in smart contracts, there are many factors that affect fuzzing, including input generation, transaction ordering, and oracle quality [42], [43].

Foundry should be viewed as an advanced engineering tool rather than a fully sound formal proof system. The primary benefit of using Foundry is its ability to test token behaviors through a large number of realistic call sequences involving minting, burning, approvals, transfers, and delegated transfers [41]. A major limitation of Foundry is that even if no counterexample has been found after conducting many tests, there are likely many other possible input combinations that have not yet been tested [42], [43]. As such, Foundry would be particularly useful as a regression-oriented assurance layer, allowing users to rapidly

identify if an ERC-20 implementation has changed in a manner that breaks previously assumed invariants. However, when attempting to formally verify the preservation of supply or allowance correctness for all symbolic input combinations, deductive verification would need to be used in conjunction with Foundry [38].

KEVM is positioned differently within the Ethereum toolset due to its ability to reason with EVM bytecode as opposed to Solidity source code [24]. The theoretical importance for deployment-level assurance follows directly from the fact that contracts executed on Ethereum do so as EVM bytecode, not as their Solidity source file [40], thus providing an appropriate starting point for the analysis of ERC20-I3. Since transfer supply preservation can be formalized as a reachability statement with respect to storage slots, including the requirement that the totalSupply slot remains unchanged, this provides an example of how source-level reasoning has limitations when compared to what is known about source-level semantics, compiler behavior, and bytecode execution [24], [40]. However, Chapter 5 also demonstrates that KEVM was the least practical tool in our experimental setting due to compilation and backend restrictions that prevented the proof from completing. While such results demonstrate no weakness in the semantic foundation of KEVM, they do illustrate that a very powerful verifier may still be impractically difficult to use in verifications at thesis or developer scale [24], [40].

In terms of SPL Token verification, the best option available using a reduced model was Kani. Seven Kani proof harnesses were written and verified, covering all six SPL Token invariants described in Chapter 4 [27]. The results are important as they show that Kani's instruction-based model closely mirrors the Solana instruction model, where each instruction has explicit account information, and therefore Kani can check valid account states up to a given bound using symbolic execution [27]. Kani achieves this by first converting Rust's mid-level intermediate representation to Goto-C and then using the CBMC bounded model checker [45] to discharge the resulting verification conditions. This process allows each instruction harness to be checked thoroughly up to their respective unwinding limits. Although Kani was an excellent choice for SPL-I3 transfer supply conservation, SPL-I4 mint authority enforcement, SPL-I5 token account mint matching, and SPL-I6 delegate amount enforcement, as these properties can be reduced to bounded symbolic checks on simplified MintAccount and TokenAccount structures [33], this success comes with a catch. The Kani model does not consider full Solana runtime behavior such as account rent, cross-program invocation, account scheduling, and full public-key semantics; thus, the result verifies the reduced instruction

model rather than the fully functional SPL Token program in its complete runtime environment [32]. This disparity is not merely a characteristic of Kani. It reflects an architectural distinction investigated by RQ3: the EVM exposes a single, shared contract state that a Solidity tool can address directly, whereas the Solana Runtime distributes the same correctness responsibility across independently held accounts, which a Rust tool can reach only through an abstracted model. The variation in tool performance between the two platforms therefore necessarily reflects the differences in execution and state modelling that RQ3 examines.

Prusti operated effectively as a specification-oriented and diagnostic tool. Its precondition and postcondition notation were consistent with how SPL Token invariants are specified through functional contracts over account fields [28], [44]. While Kani is most effective at finding violations within a given bounded search space, Prusti is most effective at identifying why a faulty implementation violated a specific SPL Token invariant. Chapter 5 shows that Prusti successfully verified the correct SPL Token implementations while falsifying the faulty implementations, specifically transfer supply modifications, missing mint authority checks, missing delegate amount reductions, and burn inconsistencies. This verification capability is essential in audit environments as it identifies the specific postcondition failure resulting from the implementation error, rather than simply reporting that an assertion could not be proved or was violated [28], [44]. This result is also consistent with the broader body of literature on verifying Rust programs using Rust's ownership and type disciplines as a basis for modularly verifying program properties [28], [46].

MIRAI extends Kani and Prusti by employing abstract interpretation of Rust's mid-level intermediate representation (MIR), which it uses to verify all six correct SPL Token functions as shown in Chapter 5. It detects the same four faulty SPL Token functions as Prusti and also identifies a reachable arithmetic overflow within an unannotated SPL Token function. This allows MIRAI to serve as an early indicator of potential runtime safety issues that may not yet have been developed into full invariant specifications [29]. The role of MIRAI is therefore distinct from that of Prusti. While Prusti provides more precise results when the expected property is provided as a contract prior to analysis, MIRAI identifies additional path-sensitive runtime risks for which no specifications yet exist [29], [44]. However, MIRAI's conservative reporting behavior is limited by the fact that it treats many caller-dependent overflow conditions as implied preconditions. As such, MIRAI could potentially withhold warnings that would need to be explicitly identified in a higher assurance proof [29].

The overall results help to answer Research Question 2. They show that the chosen tools are able to identify and check many of the critical fungible token invariants, however they do not guarantee complete assurance across all necessary aspects. Certora's strength was in source-level deductive ERC-20 rules, Foundry's strength was in regression testing of real-world invariant values, KEVM provided the strongest bytecode-level assurance in theory, Kani was most effective at verifying bounded Rust harnesses, Prusti provided the best diagnostic information for explicit contracts, and MIRAI was the most effective lightweight abstract interpretation and runtime risk detection tool [28], [29]. Therefore, the most reliable verification approach would involve combining these tools through a layered compositional verification strategy. For fungible tokens, deductive verification should be performed for core functional invariants; fuzz testing should be conducted to determine if counterexamples exist across various sequences of states; bytecode-level validation should be performed whenever deployment-level correctness is a concern; and Rust-based verification should include documentation of which Solana runtime features were abstracted away [41], [44]. A layered interpretation is more realistic than claiming full verification from a single tool, which risks providing a false sense of security. Additionally, this approach allows users to better understand the limitations of cross-platform formal verification. Beyond its practical dimension, this is also where the results of the tool-effectiveness evaluation begin to feed into the answer to RQ4. The finding that no single tool provides complete assurance is what led to the recommended approach for developers, auditors, and security engineers, presented as a series of stages in Section 6.9. The layered approach outlined above therefore provides the link between the effectiveness evaluation that addresses RQ2 and the practical applications that address RQ4.

6.2 Observations and Limitations of Tool Application

The applied *formal verification* techniques demonstrated that *formal verification* is not an entirely automatic task in which a tool merely accepts code and delivers a definitive true or false answer. Rather than being fully automated, every conclusion depends upon a series of modeling decisions. How was the *invariant* defined? Which *storage variables* or accounts were chosen to be represented? What mathematical assumptions were made for the purposes of the analysis? How did the *harness* constrain the *execution paths* available to the code? Finally, how well does the *verification model* represent the actual *runtime* behavior of the *blockchain* [27], [28]?

This observation has implications for the results generated in Chapter 5 since there were many successful verifications reported; however, they cannot be viewed as absolute or unconditional proofs of behavior exhibited by all deployed *ERC-20* or *SPL Token* implementations. Results from the experiments performed in this study can be considered valid only with respect to the specified portions of the implementations used as input to the verification process, the semantics of the tools employed to perform the verifications, and all assumptions documented in the experimental design [29].

6.2.1 Specification and Harness Dependence

A significant finding common to both environments is that tool performance depends heavily on the quality of the specifications and harnesses defined. While Certora was able to clearly specify ERC-20 supply and allowance properties in the Solidity experiments, the balance consistency rules presented difficulty when insufficiently explicit arithmetic assumptions were used [25], [37]; this did not indicate that the ERC-20 implementation was incorrect but rather that deductive verification requires an explicit definition of the intended mathematical domain [7], [38]. Similarly, Foundry demonstrated this dependency in a different way: before the handler was added, the campaign revealed weaknesses in the invariant model rather than errors within the implementation, because the sum of balances was calculated over a much smaller actor set than had been assumed [43]. As such, there are important analytical implications that arise when a tool fails to produce a passing result; a non-passing result does not simply reverse the interpretation of a passing result, i.e., that the code is correct. Likewise, an inconclusive or falsified result may identify issues with incomplete or unsound definitions of the property being specified rather than incorrect code, which is precisely the internal validity issue discussed in Section 3.8.

The same relationship appeared in the Rust-based experiments. Kani required proof harnesses that bounded symbolic input values and represented the basic SPL Token account structures [27], while Prusti required sufficient precondition and postcondition detail to define the balance, supply, authority, and delegation inter-relationships [44]. MIRAI was able to identify some runtime errors without a fully annotated specification; however, it had to determine when to report caller-dependent arithmetic issues based on the stringency level of its reporting policy [29]. Therefore, this experiment shows that successful verification depended on both the capability of the tool and the ability of the analyst to model Solana account behavior in a

way that was sufficiently simple for the verification tool to process, yet sufficiently expressive to support a meaningful finding [22].

6.2.2 Runtime Abstraction and Platform Fidelity

The findings of the Ethereum experiment reveal the gap between assurance at the Solidity source level and assurance at the EVM bytecode level. Certora and Foundry are closely aligned with the Solidity source abstraction level, making them useful for expressing ERC-20 storage properties [37]. However, as noted previously, source-level verification cannot entirely bridge the gap between Solidity code, compiled EVM bytecode, and deployed EVM bytecode [40]. KEVM addresses this gap through formal EVM semantics; however, its application to this problem was limited by operational complexities and backend constraints rather than by any weakness in its underlying semantics [24]. The experiment demonstrated a case in which a usability limitation prevented an otherwise theoretically sound verifier from producing any result, meaning that the tool capable of providing the most accurate deployment-level guarantees ultimately provided no verification output. My experience illustrates how this issue extends beyond this research, as Garavel et al.'s expert survey identified usability concerns such as installation, maintenance, and tool operation as significant obstacles to the adoption of formal methods, alongside expressiveness and scalability [47]. Therefore, usability must be considered a component of tool effectiveness, and tools that are practical to deploy are far more likely to deliver value in real verification settings.

The key constraint for Solana was not a source-to-bytecode gap but rather how the runtime behavior of accounts was abstracted. Kani, Prusti, and MIRAI all successfully reasoned about simplified Rust models of SPL Token instruction logic; however, none of them incorporated Solana runtime behaviors such as account rent, cross-program invocation, parallel transaction scheduling, or full public key and program-derived address semantics. This is significant because Solana correctness depends on whether the correct accounts have been passed, owned, signed, and written to relative to the expected mint or authority [8], [22]. Therefore, the Solana results provide strong evidence regarding instruction-level arithmetic and authorization logic but do not constitute complete proof of all potential runtime account interactions [29].

Furthermore, there is an implication that connects this section to Section 6.1: since the Ethereum tools were applied against real OpenZeppelin code whereas the Solana tools were

applied against a manually constructed model of the SPL Token account structure, what appears to be a verifiability advantage of the Rust ecosystem may simply reflect a difference in target fidelity rather than an inherent property of the tools themselves. Additionally, disclosing scope reductions under the rules of Section 3.2.2, rather than allowing them to remain implicit, transforms a hidden risk into a defined and reviewable bound; however, this does not eliminate the associated limitations on external validity.

6.2.3 Summary of Main Tool-Application Limitations

Table 17 consolidates the limitations observed during application. It does not diminish the value of the results; it clarifies the boundaries within which they should be interpreted [44].

Limitation observed	Where it appeared	Analytical implication	Possible mitigation
Arithmetic-domain assumptions	Certora, Kani, MIRAI	Proofs may depend on excluding overflow-reachable values [7], [25], [27], [29]	State arithmetic bounds explicitly in specifications and preconditions
Harness and coverage dependence	Foundry, Kani	Results depend on whether generated states represent meaningful token behavior, and a fuzzing pass is necessary but not sufficient [27], [41], [42]	Design handlers and harnesses around valid state transitions; pair fuzzing with deductive proof
Source-to-bytecode gap	Certora, Foundry	Source-level assurance does not fully prove deployed bytecode behavior [24], [37], [40]	Combine source-level tools with KEVM or bytecode-level checks
Runtime abstraction	Kani, Prusti, MIRAI	Simplified SPL models may omit Solana runtime account effects [17], [18], [22]	Document excludes runtime features and verify account checks separately
Target fidelity	Kani, Prusti, MIRAI	Verifying a reconstructed model is not directly comparable to verifying genuine source [27], [32]	Report the modeled fragments explicitly and avoid cross-ecosystem score comparison

Tool setup and resource limits	KEVM	Strong semantics may remain difficult to apply in practice [24], [40], [47]	Use reduced bytecode targets or a provisioned native verification environment
Conservative reporting	MIRAI	Some risks become implicit preconditions rather than reported warnings [29]	Use stricter diagnostic settings and explicit precondition review

Table 17: Summary of observed verification limitations, their analytical implications, and possible mitigation strategies across the selected tools

Overall, the application of tools within this thesis demonstrates how *formal verification* evidence can be interpreted across multiple abstraction layers. While all six tools, *Certora*, Foundry, KEVM, Kani, Prusti, and MIRAI, provide assurance through different aspects of analysis, each also exhibits limitations. These limitations become apparent when the same invariant categories are compared across different verification methodologies [28], [29]. This addresses RQ4: the practical implication is that no verification result should be interpreted in isolation from its specification, harness, abstraction layer, and runtime assumptions. For high-value tokens, the most robust assurance will come from using an ensemble of tools, making all assumptions explicit, and treating formal verification as a complement to testing, auditing, and security review rather than a replacement for them at the point of deployment [41].

6.3 Comparison of Token Invariants Across Ethereum and Solana

The invariant framework of Chapter 4 was built on a deliberate symmetry: six ERC-20 invariants paired one-to-one with six SPL Token invariants, each pair encoding the same correctness intent. A purely descriptive reading of Chapter 5 would conclude that this symmetry held in practice, since the conserved quantities and authorization conditions recur on both platforms. The more important finding is that identical correctness intent was discharged by fundamentally different means on each platform, and that this difference undermines any direct comparison of the verification outcomes. Because Ethereum holds token state inside contract storage [4], [11] whereas Solana distributes it across externally supplied accounts validated at runtime [5], [17], an invariant that Ethereum must establish by proof is frequently converted on Solana into a structural guarantee enforced by the language or the account model before verification begins. The consequence is that a verified result on one platform and a verified result on the other are not evidence of equivalent assurance, because they rest on different amounts of prior structural enforcement [7].

This distinction is most evident with the arithmetic invariant of conservation. ERC20-I1 and SPL-I1 both represent the same relationship between holding and total supply, however the location at which this information resides is not just a matter of coding style; it dictates how much the tool must prove. Since ERC-20 stores both balances and supply in one smart contract, the relationship exists as a single entity in the tool space, thus why the Certora failure was due to unbounded arithmetic in proving balance consistency versus an inability to relate variables [25]. SPL-I1 required two Kani harnesses, one for MintTo and one for Burn, since there was no direct inductive linkage between tokens held within accounts and mint.supply in the state, and thus had to be defined in the specification [33]. The important issue is not only the number of harnesses required, but the location of the relationship being verified.; on Solana the user supplies a relationship that Solana's storage format provides for free, therefore some of the correctness proof will move from the implementation into the proof where it is now a necessary assumption used in the proof rather than something guaranteed by the code [7], [33].

Transfer supply conservation sharpens this. ERC20-I3 and SPL-I3 conserve the same quantity, but the burden is reversed:

1. On Ethereum there is no language-level restriction in the EVM preventing transfers from accessing _totalSupply; therefore, the only way to prove these are conserved quantities is to demonstrate that no possible path exists through the EVM where a write occurs to the supply location. This is precisely what the KEVM reachability claim was designed to prove [24].
2. On Solana, the Transfer instruction references the mint account as a read-only value; therefore, at compile time Rust prevents writing to mint.supply through its ownership rules, and Kani proved it within those constraints rather than exploring all possible paths [27].

The same invariant is a valid proof obligation on one system, but nearly tautological on the other. This impacts how much weight to assign to these findings. A finding that SPL-I3 holds is weak evidence that a tool can perform its task well; because the language implements all the important constraints here . A finding that ERC20-I3 holds would have been a very good indicator of tool performance, if KEVM could be installed [24]. Finding that both are verified will conflate a structural correctness property with having proven an assertion. The authorization invariants show the greatest asymmetry, and again, the one-to-one

correspondence of Chapter 4 makes it difficult to see, because the mapping crosses index boundaries:

1. The ERC-20 has an allowance which is a relation of two addresses in a nested mapping, whereas SPL maintains the authorized amount to delegate for a particular token account it manages. Both have been formally verified; however, the ERC-20 is relational from its design while the SPL is localized, and therefore the same results mask different levels of complexity when verifying the specifications of each.
2. Authorization for changes in supply (ERC20-I5 and ERC20-I6 vs. SPL-I4) represents the greatest difference. ERC20-I6 could not be validated at all due to the fact that OpenZeppelin declared _mint as internal and placed authorization in the contract being deployed, thus moving the authorization beyond the scope of the verification target. In contrast, SPL-I4 was validated by Kani, Prusti, and MIRAI because mint_authority is a field the program itself is required to read. The disparity here is not that Solana is more secure than Ethereum but rather that the decision to authorize is made externally to the artifact being analyzed within the context of the ERC-20 standard library.
3. Account validity (SPL-I5, with no ERC-20 counterpart): the absence of a counterpart is itself the analytical finding. ERC-20 cannot express this invariant because a caller cannot substitute a foreign token account, so the obligation never arises. Solana's account-passing model reintroduces it as a first-class correctness requirement [22], [32].

All three follow an identical pattern. On one hand Ethereum places token state into the contract and thus allows arithmetic to be checked directly as part of verification but pushes authorization into the world outside of the contract. On the other hand, Solana distributes token state over explicitly passed accounts and therefore allows authorization and account identities to be checked internally as part of verification, however it requires analysts to reassemble relationships that the EVM would have been able to do automatically. As such, neither system has a uniform advantage when it comes to verifying invariants; rather each architecture makes different subsets of invariants easier while making others more dependent on assumptions. Similarly, the vulnerabilities associated with both systems reflect this division, with Ethereum failures tending toward arithmetic and supply control errors while Solana failures tend toward account validation and authorization errors [32].

One major takeaway is that there is symmetry to this invariant framework in terms of intentions; however, the symmetry completely disappears while examining the results more critically. Chapter 5's verification results cannot be used for a head-to-head comparison of which platform provides stronger correctness guarantees, because the same label represents different amounts of proof and different structural representations for each platform. Additionally, as noted in Section 6.2 regarding target fidelity, the apparent verification advantage of the Rust platform reflects two aspects: how well the language and account model represent obligations prior to running the tool, and how close the Solana targets were to actual production code versus reconstructed models [32]. This answers RQ3 that any legitimate cross-platform claims must include not just whether an invariant was proven, but how much of the correctness came from the language and account model versus the actual proof [7], [24].

6.4 Impact of EVM and Solana Runtime on Invariant Design

Section 6.3 illustrated that invariant symmetry exists only for correctness intent. The remainder of this section illustrates that the EVM and Solana Runtime do not simply differ on how they run token logic. They both also determine what could constitute an invariant. A given invariant has to be located within some part of the platform's state model, it has to be governed by an execution rule, and it has to be defined through a verification boundary. In other words, the EVM defines invariants primarily over the state space owned by each contract, while the Solana Runtime defines invariants over account relationship information, signer authority, and mutability constraint data [17], [18]. Both runtimes define where the line is drawn between what will be enforced during execution and what must be proven separately, which mirrors Schneider's definition of properties that can be enforced using execution mechanisms versus those that must be reasoned about separately [52]. As discussed in RQ3, since that line is predetermined prior to the selection of any tools, invariant development was already shaped by the specific platform from the outset.

The token contract is the primary unit for reasoning within the EVM model. ERC-20 balance information, allowance data, and total supply values reside in storage with an easily deterministic arrangement. Consequently, the verifier has a straightforward relationship with respect to correctness, specifically balance consistency and transfer supply conservation, based solely on the states of the contract itself [48], although these relationships appear compact due to their internal nature, compactness does not necessarily imply simplicity. Due

to the fact that the EVM provides unbounded capabilities to allow any arbitrary code to read from or write into storage, it is necessary to verify that there exists no possible execution path which would modify protected state in an invalid manner. ERC20-I3 holds true not because a method is named transfer, however only after demonstrating transfer will never alter _totalSupply on any viable path [24], [25]. While the EVM models storage locally, it places upon the verification effort the responsibility for ensuring that no bad writes occur. Gas accounting exacerbates this by adding additional complexity to the problem, and gas accounting was abstracted in Chapter 5. Since the EVM's execution model includes gas accounting [49], gas exhaustion could cause execution to be reverted and create vulnerability patterns that source-level analysis might not identify directly [55].

Solana reverses this. The runtime requires all transactions prior to execution to list an account, with read/write status, of all signers, and their permission for each account they will be accessing [19]. In addition, passing the mint account as read-only restricts the scope of the supply preservation problem since the model already limits what data can be changed and therefore the verification tool does not need to examine all possible changes to supply [27], [32]. Instead of proving correctness at runtime via proof, this correctness is enforced by construction [52]. Languages like Move take this further by implementing asset-based models where an asset becomes a resource that cannot be duplicated or lost inadvertently [53]. Since SPL Token does not use such a resource-type based model, the verifier must manually verify that the accounts are being correctly used, are owned by the expected program, are associated with the correct mint account, and have been properly signed by the expected signer [32], [50]; restricting writes does not make establishing Solana invariants simpler in general, but instead adds another requirement to the verifier, namely, verifying the accounts passed to a transaction are valid.

This creates a new definition for an already-verified property. On Ethereum, this typically signifies that a method retains a relationship within storage under all modeled possible execution paths. In contrast, Solana can signify that an instruction's logic will preserve a relationship given a valid account configuration. The latter condition is contingent upon additional factors, valid ownership, mint matching, and signer status, being assumed in order to confirm the validity of part of the arithmetic transitions [32]. Therefore, simply based upon the fact that a Solana verification appears to be more robust than its Ethereum counterpart solely due to the shift of proof obligations into preconditions; the point is further emphasized through the abstraction of the parallel execution model in Chapter 5 [5], [19]. An invariant

confirmed against a sequentialized harness does not provide information regarding the schedule interleaving permitted at runtime as regards a shared token account; therefore, this represents the greatest fidelity for the design where the runtime is most distinguishable [8], [19].

Beginning with the EVM's weakness, it is not the lack of structure that limits the ability to ensure correctness for each contract logic; it is the fact that all relevant relationships in an ERC-20 contract are established by implementation discipline, meaning developers implement the necessary storage updates [48]. A single error in updating storage or checking access control will violate the invariant as the runtime has no means to differentiate token state from any other storage [24]. Solana's weakness is the opposite: explicitly defined account identities create a new specification burden. This is demonstrated by SPL-I5 as there is no ERC-20 equivalent, since callers on Ethereum do not have a mechanism to replace a foreign token account; however, token accounts in Solana must be externally provided and verified [22], [50]. While the difficulty in establishing these specifications is not eliminated, it is merely relocated from hidden relationships between contracts based on storage to explicitly defined relationships between accounts; failure to establish even one of these relationships may result in an unsound proof, and this is the defect class most commonly reported in symbolic execution studies of Solana [32], [51].

In essence, it is a fundamental principle of the runtime environment to determine which invariants can be established at little or no cost versus those that incur significant costs. For example, Ethereum provides mechanisms for making storage relationships inexpensive to locate but costly to establish as true for any path. On the other hand, Solana has mechanisms for exposing authority information inexpensively but establishing faithful models of such authority is costly. Thus, a cross-platform verification framework cannot simply assess whether ERC20-I3 and SPL-I3 have been verified, but rather what the runtime provided prior to the tool's process beginning, what the tool proved during its process, and what assumptions were made in the specification [27], [32].

6.5 Impact of Execution and State Models on Specification

The nature of execution and state models not only influences which invariants will be chosen, but also how they are represented. An informal property such as "transfer preserves supply" can be formalized in terms of concrete variables, accounts, preconditions, and postconditions following the specification grammar of Hoare logic [25], [54]. The experiments demonstrate

that the greatest divergence among the three platforms exists at the translation step from a given property into a specific specification language. While all three tools use the same vocabulary for describing correctness, they do not use the same grammar for specifying behavior. The primary assertion in this thesis is therefore that specification effort depends on the pairing between a tool and a particular state model, not solely on the characteristics of the tool itself.

6.5.1 Ethereum: the contract as specification boundary

In the case of Ethereum, the boundary of the system is defined within a single smart contract. Many of the ERC-20 properties are therefore directly stated through the state model: balances and allowances are updated, and _totalSupply remains unchanged unless there is a call to the minting or burning function [48]. Both Certora and Foundry specify these as contract-level rules. However, the source-level specifications are based upon Solidity variables, whereas deployed behavior is represented in EVM bytecode [24], [49]. There is thus a gap: proving that the Solidity representation maintains an invariant is not equivalent to proving that the compiled bytecode maintains an invariant according to formal EVM semantics [24], [40], [49]. KEVM bridges this gap; however, the experiment demonstrated that there is a practical cost associated with the increased precision inherent in verifying bytecode-level versus source-level rules [40]. Thus, Ethereum's specification challenge is not only how to write an invariant but at which level of abstraction it should be established.

6.5.2 Solana: specification as runtime reconstruction

Solana's specification challenges stem not from source-to-bytecode abstraction differences but from runtime reconstruction. All three tools, Kani, Prusti, and MIRAI, reason over Rust; however, correctness relies on several factors including ownership, signer status, writable flags, mint associations, and PDA derivations [50]. Some of these factors cannot be adequately represented by a small Rust function alone. It is necessary to model these factors in some form to produce a complete specification; alternatively, they may be included as preconditions. If the specification oversimplifies the account environment, a proof that was valid within the harness provides little assurance regarding the correctness of the deployed application [51]. As such, Solana specifications become larger even though the invariants themselves appear simpler. A transfer conservation specification must provide information about which token accounts are being compared, whether the accounts share a mint, and

whether the signer or delegate owns the source [50]. While Ethereum encodes these factors into contract storage, Solana reconstructs them at the specification boundary. The same asymmetry appears in specifying frames. Frames are just as important as transitions for stating what remains unchanged; however, specifying frames requires a clear understanding of what must not change [56]. In Ethereum's case the frame is contract storage, while on Solana it is an open-ended list of provided accounts.

6.5.3 Comparability and specification discipline

Specification effort is therefore not directly comparable across platforms. On Ethereum it is spent establishing storage relations and ensuring that no unintended writes occur; on Solana it is spent establishing valid account configurations and ensuring that account substitution does not occur. A short Solana proof is not automatically stronger than a long one; similarly, a failed *Ethereum* proof is not automatically weaker than one that succeeded. What a tool proves is determined by what its execution model has already established. Since *Rust* contains unsigned types, non-negative amount *invariants* are nearly trivial to state; account validation *invariants* remain difficult to establish [32]. The expressiveness ratings described in Section 3.7 do not completely reflect this, because a property can be formally expressible while simultaneously reducing what the proof establishes due to silent execution model implications. Each single-instruction specification on *Solana* also implicitly contains an unstated assumption of non-interference regarding parallel *runtime* behavior [19], [27].

Execution models can also replicate platform assumptions rather than testing them. This occurs when a specification assumes that a single function is responsible for modifying a piece of state when there may be multiple paths to modify that state via inherited hooks or internal functions. Similarly, this can occur on *Solana* when a *harness* assumes that a *token account* corresponds with a *mint*, or that an authority has signed, when this correspondence must be verified by every program [51]. In both cases, the specification ends up representing the designer's intentions rather than demonstrating how an adversarial execution might cause an *invariant* to fail [31].

These results suggest a multi-layered view of specification. Robust specification on *Ethereum* consists of combining source-level rules with bytecode-level checks wherever possible, particularly for supply and *authorization* [49]. Robust specification on *Solana* consists of separating arithmetic transition properties from account validity properties so that proofs do not quietly depend on unverified assumptions about account relationships [50]. This approach

is more informative than a single *verified* label because it demonstrates whether the proof addresses implementation logic, *runtime* assumptions, account relationships, or only a reduced model. The conclusion is not that one execution model suits *formal verification* better than another, but that each demands a different specification discipline, and cross-platform assertions are meaningful only after those differences have been explicitly identified [32].

6.6 Tool Capability Comparison Across Solidity and Rust Ecosystems

The tools should not be evaluated on how many invariants each tool verifies. One tool could prove a small property with strong assumptions and appear more effective than another that identifies potential ambiguity within the definition of the system being modeled. Therefore, capability is defined as the connection between how a tool performs verification, the structure of the invariant, and the degree of abstraction at which it works. And this must be done with consideration for the confounding variable from Section 6.2. The Solidity tools are applied to real OpenZeppelin code, whereas the Rust tools are applied to constructed account models. Therefore, any perceived difference in capabilities of the tools will at least partially represent differences in the fidelity of their application targets versus differences in the functionality of the tools [32].

The most supportive system of explicit token-level specification in the Solidity environment was provided by Certora, since the properties of an ERC-20, such as the preservation of supply and the decrease of allowance, can be explicitly defined over the contract's state storage variables [25]. However, at the same time these results show the limitation of deductive verification: if it cannot prove something, this does not necessarily mean there is anything incorrect about its implementation. For example, the balance consistency case was blocked by the difficulty of modeling arithmetic instead of having proven some fault; therefore, Certora is both strong and unforgiving, and in order to obtain good results the problem must be specified precisely, otherwise no useful answers will be produced [7]. On the other hand, Foundry showed an inverse profile. It was friendly to developers and practical for exploring execution sequences; however, a successful invariant test only means that Foundry has not been able to find any counterexamples during the course of the campaign and proves nothing else [26]. Therefore, its main use is as a diagnostic tool, as Foundry often reveals errors in the harness before formal proof begins.

The Rust tools followed a different path. Kani was successful in testing bounded behavior of isolated SPL Token logic with harnesses and symbolic inputs; Prusti had an advantage when

there were specific preconditions and postconditions to define, with clear contract definitions and more accurate diagnostics; MIRAI added another layer of value by finding reachable panics via abstract interpretation without needing a complete invariant [29]. Together, these provided broader diagnostic coverage than any one of the Solidity tools; however, it does not mean their coverage is more assured, as most of the coverage came from simplifications of how accounts in Solana behaved. Table 18 summarizes the strongest capability and main limitation of each selected verification tool.

Tool	Strongest capability	Main limitation
Certora	Contract-level rules for ERC-20 storage invariants	Sensitive to arithmetic and specification assumptions [25]
Foundry	Practical invariant fuzzing and counterexample discovery	A passing result is not a proof [26]
KEVM	Bytecode-level semantic assurance	High setup and usability cost [24]
Kani	Exhaustive bounded checking of Rust logic	Bound and harness dependent [27]
Prusti	Precise pre/postcondition reasoning	Requires annotation and simplified targets [28]
MIRAI	Panic and reachability diagnostics	May turn risks into implicit preconditions [29]

Table 18: Comparative summary of the strongest capabilities and main limitations of the selected formal verification tools

The most important finding was how each ecosystem failed in fundamentally different ways. Tools using Solidity struggled with the differences between a developer's source code level description of what a smart contract is supposed to do, the specification, versus the bytecode level of the contract that the EVM will execute; whereas tools for Rust struggled with the difference between a single function written as a Rust program versus the total runtime behavior of a Solana program. Therefore, we cannot naively conclude that Rust tools verified more and therefore Rust tooling is better. The Rust tools were able to verify properties within

smaller constructed models of the smart contracts. In contrast the Solidity tools were closer to the actual state of the contracts but had to contend with much more open-ended storage behaviors. Therefore, capability is complementary, not hierarchical. The tools can be used together to get evidence of correctness in combination rather than selecting a single tool, and this addresses part of Research Question 2 without creating a ranked list [7].

6.7 Verification Effort and Developer Usability

The verification effort was primarily based on the difference between how developers think, using operations, and what the tools require, with specific statements. Developers are thinking of operations like transfer, mint, and burn. Tools are looking for descriptions of data stored in variables, an account field, symbolic inputs, possible overflows, and postconditions. That is where the bulk of the effort to verify smart contracts lies. And this difference varied significantly between the two systems.

Certora offered a clear path from an invariant to a rule for auditing Solidity code once all of the appropriate storage variables had been identified. However, its usability decreased dramatically if a property rested on some sort of very delicate arithmetic assumption. In this balance consistency example, the difficulty in defining the rule was less about defining the rule itself but about defining the correct mathematical context, illustrating that a user-friendly language could be used to create difficult-to-verify properties. Foundry's integration into the typical testing flow made it easier to implement, but the lower assurance it provides means it should be considered a first layer tool that validates the harness and identifies counterexamples prior to attempting a potentially much more costly proof process. KEVM was by far the most difficult to implement, having the lowest usability, because its similarity to EVM semantics is precisely what makes KEVM so demanding. It requires at a minimum bytecode level reasoning as well as a specialized setup environment. Therefore, the issue with KEVM is one of accessibility and not one of capability.

The Rust experience required significant effort to build runtime models and construct harnesses. Unlike the Solidity tools, which operated closer to the actual contract state, Kani needed bounded harnesses, Prusti needed explicit contract definitions, and MIRAI needed to interpret warnings as well as include implied preconditions [27], [29]. This additional effort arose because Solana correct functioning depends upon account configuration and is therefore not fully represented by a small Rust function, which means the analyst has to determine what should be modeled, assumed, or excluded. This creates a serious usability concern: even

though a harness looks simple, there could be many significant assumptions made about ownership, valid signers, and similar factors, such that when the developer receives a Verified result they do not see how conditional that result is. As a result, usability is not just how easy it is to run a formal verification tool, but how transparently that tool reveals the assumptions underlying its result [27].

These qualitative results are supported by objective measures from the campaigns, providing a quantifiable rather than subjective basis for comparison. The Certora specification comprised nine CVL rules spanning the six ERC-20 invariants, of which four were Verified and five Inconclusive [25]. The Foundry campaign issued 128,000 calls per invariant across five handler functions, with the full campaign completing in 25.61 seconds [26]. The Kani suite used seven proof harnesses to discharge all 597 checks in approximately 1.07 seconds [27]. MIRAI analysed twelve functions in the crate and emitted only five diagnostics [29]. Prusti detected all four seeded faults, though at substantially higher annotation effort per function [28]. KEVM, by contrast, produced no measurable verification time, since the cost of configuring its environment prevented even a single reachability claim from executing [24]; this records its poor usability as an observed outcome rather than a subjective judgement.

The following multi-stage process is therefore recommended instead of relying on a single method:

1. First, conduct invariant fuzzing and testing. At this initial stage, Foundry is particularly effective as it can easily identify errors within the harness, test its bounds, and determine whether the invariant has been properly modeled [26].

2. At the second stage, use bounded and deductive verification to verify critical invariants. Supply preservation, delegated authorization, and valid Solana account checks all have properties that can only be proven through more rigorous methods of reasoning than fuzzing alone can offer. Tools such as Certora, Kani, and Prusti are most suitable at this stage.

3. Bytecode-level and runtime-aware checks should only be performed when greater deployment assurance is required. KEVM-style verification is more costly but provides a better representation of deployed EVM behavior versus source-level Solidity behavior alone.

4. Finally, clearly document the limitations of each result. Any verified result should clearly indicate what assumptions were made, what was excluded, and how much effort was required from the tools, as formal verification is only practically useful when its boundaries are visible [47].

6.8 Theoretical Implications

This thesis makes four contributions to the academic understanding of token correctness and cross-platform formal verification.

The symmetric invariant framework presented in Chapter 4 is the first contribution of this thesis. Most earlier contributions treat token standards either as interface specifications or as ecosystem components. Similarly, surveys of verification tools tend to focus on the Ethereum ecosystem alone [72], and formal specification surveys highlight the scarcity of cross-ecosystem treatments [68]. Rather than treating each platform independently, this thesis defines four correctness families, balance consistency, supply preservation with minting constraints, transfer validity, and delegation and authorization, derived from platform-independent correctness requirements. Each family is then refined into two concrete, one-to-one invariant sets, namely ERC20-I1 through ERC20-I6 and SPL-I1 through SPL-I6. The ERC-20 set is defined over the storage variables of the OpenZeppelin implementation [25], whereas the SPL Token set is defined over the mint account and token account fields of the SPL Token program [17]; both sets are grounded in the interface obligations defined by the ERC-20 standard [13]. The advantage of this framework is that it is reusable and structural: it establishes a common vocabulary in which an obligation specified by the ERC-20 standard and its SPL Token counterpart can be paired and reasoned about together under the same structure, rather than treated as separate artifacts belonging to different execution platforms.

The second contribution of this thesis is the empirical work resulting from testing six formal verification tools using same invariant framework on both Solidity and Rust. The six tools tested were Certora, Foundry, and KEVM for Solidity, and Kani, Prusti, and MIRAI for Rust. The results documented in Chapter 5 and summarized in Table 16 show not just what invariants each tool can prove, but also under what conditions and assumptions they do so. All results are documented with information about the tool used, its version, the corresponding compiler versions, and the versions of the libraries. There therefore exists an empirically verifiable basis for comparison that allows the quality of individual tools to be assessed across different ecosystems [72].

Thirdly, there is a conceptual contribution that forms the core theoretical statement of the thesis; verified labels are platform dependent. The same correctness intention can impose different proof burdens depending on where the constrained state resides within the platform's architecture. Bhargavan et al. identify the gap between source-level proofs and full bytecode-level verification coverage as a fundamental issue in smart contract correctness [7]. An obligation imposed on Ethereum in terms of establishing proof is often translated into a structural constraint within Rust's ownership model and the account passing runtime before formal verification begins [32]. For example, Navas demonstrates that the token account to mint relationship represents the principal structural constraint supporting the supply proofs in the formal verification of SPL Token 2022 [33]. Thus, ERC20-I3 requires a reachability proof showing that no execution path writes to total supply, whereas SPL-I3 is nearly tautological because the mint account can be passed as a read-only reference [24]. A further extreme case is represented by SPL-I5, which has no ERC-20 analogue at all [22]. This thesis therefore establishes how cross-platform verification claims should be constructed, since assurance cannot be measured solely by comparing the number of verified invariants.

The final contribution of the thesis relates to methodology. In this respect, the thesis introduces the concept of target fidelity, which refers to applying Ethereum tools to actual OpenZeppelin source code and applying Solana tools to reconstructed account models. Target fidelity is identified in the thesis as a source of variance in cross-ecosystem comparisons and should therefore be clearly stated rather than concealed. The thesis applies this discipline by refusing to convert different analytical findings into a single composite score for comparison. The staged assurance process depicted in Table 19 provides a multi-layered artifact grounded in these results. The artifact serves two purposes: first, it connects academic verification of tokens with the repeated finding of expert surveys that even very strong semantic tools are difficult to use in practice [47]; second, it responds to evidence showing that current security tools address only a small proportion of all high-impact failures observed in practice [57].

6.9 Practical Recommendation for Developers, Auditors, and Security Engineers

The applied value of these experiments is not that there is one better platform or one better verification tool. In fact, the more applicable finding is that the verification of tokens should be viewed as an engineering process depending on the state model of the platform, the level of abstraction for the verification tools used, and the security position of those who will rely

upon the results. For development teams, auditors, and security analysts, it is not merely important to apply formal verification tools, but to understand clearly what has been modeled, what assumptions were made when modeling the application, and what was excluded from consideration by the models. These findings are consistent with the broader body of research regarding the applicability of various forms of security tooling, such as automated analysis, where Chaliasos et al. conclude that automated analysis could have potentially mitigated fewer than ten percent of high-impact attacks against real-world systems [57]. Further, the same study indicates that around fifty percent of respondents utilize tooling capable of logical level error analysis, precisely the type of error which Bhargavan et al. contend can only be effectively addressed through formal verification due to testing's inability to evaluate all possible states of a system [7]. As such, the invariants defined in Chapter 4 represent the types of failures described as broken balance, supply, and authorization relationships, identified by Atzei et al. as responsible for producing the greatest number of financially damaging contract-level faults [15].

6.9.1 Implications for developers

The first implication is that developers need to consider how to implement token logic with respect to invariants at the beginning of the verification process. The experiments demonstrated that properties such as supply conservation and allowance reduction can be specified relatively directly when the storage variables representing each part of the state are clearly identified. Certora successfully proved the supply rules for the transfer and transferFrom functions of the OpenZeppelin contract [25], as well as both the finite and infinite allowance cases. However, the balance consistency rules produced inconclusive results in Certora, demonstrating that the simple interface presented by the ERC-20 standard is more complex than it appears [13]; a familiar property becomes fragile when the underlying arithmetic assumptions are not explicitly stated, and the verifier treats an unbounded overflow path as a valid counterexample. Developers should therefore regard invariants as explicit design artifacts whose intended mathematical domain is stated upfront, rather than as arbitrary tests applied after implementation [7].

A second developer-related implication concerns how developers build their harnesses. The experiments demonstrated that building harnesses is a first-class design task. Prior to the introduction of the handler, the naive Foundry specification generated many false positives; this occurred because the sums of balances were calculated based on a fixed set of actors

rather than all accounts that could receive tokens [26]. This was a specification error rather than a coding error, and similar patterns at scale have been reported by Qiao and Paul, who stated that the primary obstacles to fuzz testing adoption are usability issues including how a developer emulates blockchain behavior and lack of documentation; both of these issues cause developers to incorrectly configure harnesses [59]. The practical takeaway from the research results is that when developers test with an early version of their fuzz testing campaign, they will identify issues with their invariant model before learning anything about the correctness of the code being tested. A Verified result should therefore be interpreted as evidence that the harness has been correctly configured, rather than as evidence of code correctness [57].

The practical implications differ significantly for Solana developers. A number of experiments demonstrated that SPL Token arithmetic transition verification can be conducted using reduced Rust models when the account structure is reconstructed in the harness [27], and this was further validated by both Prusti and MIRAI through the addition of diagnostic capability and abstract interpretation at the function level respectively [28]. For Solana developers, the practical lesson is that business logic and account validation should not be treated as separate concerns. Account validation encompasses mint identity verification, owner verification, signatory verification, and delegate permissions, and each of these is implicated in every transfer operation. Furthermore, Solana's security documentation repeatedly identifies the absence of all four types of checks above as a recurring risk category [22]. This is most evident in SPL-I5, which has no ERC-20 counterpart for token account mint ownership; the Neodyme audit team independently identified account confusion and unverified invoked program identities as two of the most commonly encountered practical hazards in production-deployed Solana applications [64]. Adopting secure-by-design models early in development will significantly reduce the residual vulnerabilities that the verification process must subsequently address [73].

6.9.2 Implications for auditors

For auditors, the principal conclusion is that verified labels alone are insufficient. Auditors need to ask what a verified label mean in terms of the invariant being checked; which model was used; which runtime assumptions were made; and which behaviors were excluded. In this thesis, Foundry determined that all five ERC-20 invariants tested were confirmed, but this was based upon the call sequences generated and the handlers used to generate those calls

[26]. Certora provided stronger deductive evidence for several properties but left others inconclusive due to arithmetic modeling issues and the limited scope of functions that could be handled [25]. These implications are consistent with Feng et al.'s study, which found a high rate of incorrect interpretation of audit output by its intended recipients, suggesting that audit results should be viewed as evidence to be interpreted rather than as a final certification of system security [58].

The Rust verification process uses several tools, Prusti, MIRAI, and Kani, as complementary sources of evidence. Each of the four seeded errors was localized to the exact failed postcondition and source code line by Prusti, which is precisely the kind of diagnostic information auditors need to communicate to developers so that specific faults can be addressed [28]. MIRAI identified the same four errors using abstract interpretation and additionally located a reachable arithmetic overflow without requiring a specification [29], while Kani proved the correctness of all paths within its bounds [27]. Communication has been cited as a problem in DeFi auditing by Feng et al., who state that while DeFi auditing is effective at identifying logical and interaction issues, problems persist with communication, transparency, and the resolution of identified issues [58]. It follows that best practice would involve combining a tool that provides complete path-level correctness results with a tool that provides precise contract-level diagnostics, thereby supplying two important components: what needs to be fixed and where [69].

Auditors need to consider that automated tools can only discover a fraction of the defects in the area of concern. In their analysis of results from professional audits, Groce et al. report that the majority of severe errors found in those audits consisted of issues with data validation, the way that programs verify data, with access control, how a program controls what users have access to, and race conditions, when two or more operations occur simultaneously [61]. They also note that approximately fifty percent of all findings made during these audits could not have been identified by an automated tool. The focus of this invariant work aligns precisely with this observation; the data validation and access control categories addressed in the supply and authorization invariants of Chapter 4 represent some of the primary causes of errors in smart contract code, as identified by Atzei et al. [15]. Additionally, the remaining audit findings required the same type of manual judgment that no automated tool can replace.

This has direct implications for how audit reports express formal verification results. Rather than simply passing or failing an audit, reports should state the proof boundary explicitly. For Ethereum smart contracts, this means distinguishing between source-level verification, the level at which Certora and Foundry operate, and bytecode-level assurance [26]. KEVM attempts to address this distinction more directly; however, the results from Section 5.1.3 show that obtaining this assurance carries a significantly higher setup cost than using source-level tools [24]. For Solana auditors, the proof boundary must be described differently, since the concern is not only whether arithmetic transitions were correctly executed but also whether account validity checks were included. A harness that assumes a correct mint, signer, or owner without explicitly modeling those relationships omits precisely the error class that Solana's security guidance identifies as most common [22]. A proof of SPL-I3 supply conservation should therefore not be taken as evidence of SPL-I5 token account mint ownership unless that relationship has been explicitly modeled, a distinction made by the Neodyme pitfalls catalog for the same reasons [64].

6.9.3 Implications for security engineers

Security engineers accountable for an assurance strategy need to understand that toolchains should be organized as layers rather than as a single tool. *Foundry* integrates directly into the developer workflow and will rapidly expose both *harness* configuration errors and *state transition* errors; however, it is not a proof system [26]. *Certora* can provide stronger assurances for explicitly stated functional properties; however, its sensitivity to specification quality may reduce their value [25]. *Prusti* provides similar contract-level rigor for *Rust*, but with more precise diagnostic information [28]. *Kani* provides *bounded* symbolic assurance, which has limitations based on the completeness of the *harness* used, and also assumes that the values used by the *SPL-I1 harnesses* do not exceed `u64::MAX / 2` [27]. *MIRAI* provides valuable assistance in identifying reachable failure conditions and implied *preconditions*; however, its output must be interpreted with caution [29]. *KEVM* provides the strongest semantic basis for *EVM bytecode*, but at a significant cost to usability and accessibility [24].

Taken collectively, these tools enable a multi-stage verification process rather than a single verification step, with each tool addressing a distinct aspect of correctness.

Security engineers must understand that the same invariants used during development can also be applied at runtime. Chen et al. have shown that invariants related to supply, balance, and access control, as described in Chapter 4, can detect anomalies during runtime and thus

prevent transactions from proceeding incorrectly [60]. This is consistent with the finding from MIRAI discussed in Section 5.2.3, where MIRAI identified the potential for an arithmetic overflow in spl_naive_credit without any annotations present, demonstrating how a runtime monitor could identify failures prior to the development of a complete formal specification.

6.9.4 A layered assurance pipeline

The proposed pipeline for the practice of formal verification is outlined in Table 19. This pipeline provides both a recommended workflow and an interpretive framework to guide its application. Developers should avoid treating fuzz testing results as proof; auditors should avoid treating reconstructed proof harnesses as a form of complete runtime verification; and security engineers should be careful not to confuse correctness at the source level with assurance provided at the point of deployment.

Stage	**Main purpose**	**Suitable tools**	**Practical risk if skipped**
Testing and fuzzing	Expose obvious state transition and harness errors	*Foundry*, ordinary unit tests	Specifications may be built on incorrect assumptions
Source level proof	Verify core token invariants over implementation logic	*Certora*, *Prusti*, *Kani*	Critical properties may remain only informally tested
Account and runtime validation	Check Solana account identity, ownership, signer, and PDA assumptions	*Kani*, *Prusti*, *MIRAI*, manual Solana review	Proof may assume away the real vulnerability
Bytecode and deployment level checking	Increase assurance for deployed EVM behavior	*KEVM* or equivalent semantic tools	Source level results may not fully reflect deployed code
Manual audit and checklist review	Interpret assumptions, privilege paths, and integration risks	Auditor review, SCSVS and	Tool results may be over trusted or misreported

		EthTrust style checklists	

Table 19: Staged assurance pipeline for cross platform token verification

An evaluation process of this kind represents current best practice. In particular, both the OWASP Smart Contract Security Verification Standard (OWASP SCSVS) [62], with its assurance levels organized around verification techniques, and the EEA EthTrust Security Levels Specification [63], which distinguishes automated checks from manual audits and from a full logic review, have identified the need to document the scope of a certification, that is, the boundaries within which it was evaluated. Feng et al. have shown in practice that this documentation has been absent in many cases [58]. The implications are therefore as follows: developers and auditors should carefully examine how authorizations are implemented in their smart contracts. To demonstrate compliance with ERC-20, all possible pathways to invoke a mint operation must be analyzed. The inability to evaluate ERC20-I6 against the base ERC20.sol contract arose because _mint is defined as an internal function and is therefore only accessible through the deployer contract [25]. A proof over a library is consequently a component-level result rather than a system-level result; minting and burning authorities must therefore be validated at the composition boundary of inheritance, role assignments, and deployment configurations, precisely the access control area where Atzei et al. identify behavior that is frequently exploitable [15].

The corresponding implication for Solana is that account validation must be considered part of what the invariant preserves. A token program cannot be deemed valid solely on the basis of whether the arithmetic performed by Transfer, MintTo, or Burn is correct; it can still fail to prevent an attack due to wrong accounts, wrong signers, or wrong delegates [22], and program-derived address validation provides the link to that identity [50]. The Neodyme catalog documents this ordering and states that account and signer validation must be performed before executing balance logic [64].

In terms of assurance strategy, the final implication of these results is that assurance evidence must be recorded as an artifact that includes the invariant, the tool and its version, the model, the assumptions made, and the runtime behaviors that were excluded [27]. This is necessary because a verified result for ERC20-I3 does not carry the same meaning as a verified result for SPL-I3, given that the two architectures place the burden of correctness at different points across implementation, runtime, and harness [7]. A cross-platform assurance report must

therefore describe the assurance boundary for each platform separately rather than comparing them directly, so that formal verification becomes a discipline within token engineering rather than a certification step applied after deployment.

6.10 Threats to Validity

The threats to the experimental design are those identified in Section 3.8, which were defined before the results were obtained. This section discusses validity in the context of the results presented in Chapter 5 and interpreted in chapter 6 using the case study categories suggested by Runeson and Höst [65]. While no experimental failures occurred, successful results may still be misinterpreted, since Chapter 6 demonstrated that the interpretation of a verified outcome varies significantly depending on the tool, platform, and modeling boundaries applied. Verifying a rule in Certora, finding that an invariant passes in Foundry, proving a function with Kani's proof harness, and finding a warning from MIRAI each represent different types of evidence. Treating them as directly comparable overstates the comparability of these results [27].

Construct Validity: Construct validity refers to whether the measured variables relate to the concept that the research study purports to measure. Target fidelity (see Section 6.2, Section 6.3, and Section 6.6) was found to be the most significant threat to construct validity in this study. The tools developed for Ethereum were applied to the actual OpenZeppelin source code, whereas the tools used for Solana were applied to reconstructed MintAccount and TokenAccount models rather than to the deployed SPL Token program in its runtime environment. Therefore, when referring to a verified outcome, the researcher must recognize that the same construct was not being measured across both platforms. In one case, the experiment measured the property of compiled production code, while in the other, it measured properties of an abstract model. Sjøberg and Bergersen argue that a mismatch between the operational definition of a variable and the conceptual definition of what should be measured is the fundamental construct validity issue in empirical studies of software development. This makes construct validity arguably much more difficult to assess than other forms of validity [67]. This threat was addressed to some extent by avoiding the conversion of results from both platforms into a single score that could be interpreted uniformly across Ethereum and Solana. However, this decision cannot completely mitigate the threat, since the underlying verification targets remain different.

Modeling and runtime fidelity: The majority of the Solidity experiments operated at the contract source level; however, deployed Ethereum behavior occurs at the EVM bytecode level. KEVM was selected to bridge this semantic gap, but the significant setup cost of KEVM limited how much bytecode-level verification could be incorporated into the workflow [24]. On the Solana side, the Rust tools operated over functions and reconstructed account models but did not incorporate all elements of the full runtime. Specifically, the tools did not include representations for gas behavior, account rent, cross-program invocation, parallel scheduling, or deployment configurations [27]. The results should therefore be understood as demonstrating that certain logic preserves certain invariants within the modeled scope; they do not constitute proofs of actual deployed platform behavior, which is a limitation similarly identified in other symbolic execution studies of Solana programs [51].

Internal validity and cross platform equivalence: The interpretations made in Sections 6.1 and 6.3 explain why Certora produced inconclusive results with respect to a missing arithmetic assumption rather than an implementation flaw, and why SPL-I3 could be verified smoothly through Rust ownership rules rather than through an exhaustive proof. Both are causal claims, and a possible alternative explanation, namely a minor analytical specification error, cannot be ruled out for each individual result. A further potential threat is false symmetry: ERC20-I3 and SPL-I3 each represent supply preservation, but Ethereum requires evidence of this through contract storage while Solana's proof burden is based upon account mutability, runtime access control, and related mechanisms. Furthermore, SPL-I5 has no corresponding ERC-20 version because it represents an obligation that arises from the account passing model and does not exist in the same form under the ERC-20 standard [32]. The use of the same invariant label may therefore mask different proof burdens. This threat was mitigated by analyzing the combined results qualitatively rather than ranking the platforms numerically.

Tool maturity and researcher interpretation: The six tools represent different validation paradigms and, while they differ in maturity, automation, and proximity to deployment-level semantics, some inconclusive results may reflect limitations of the tools themselves rather than actual defects in the implementations. Additionally, some results were based upon simplified proof harnesses, and the comparison should therefore be viewed as an assurance-oriented assessment rather than a controlled comparison of equivalent verification engines [26]. The evaluation criteria described above are also somewhat qualitative and were applied by a single researcher. Runeson and Höst note that the use of multiple researchers for

classification can introduce reliability risks in case studies [65]; a second researcher might have classified a reduced verification scope as Partially Supported rather than Not Supported, as was the rating applied in this research.

External validity: The research considered specific reference implementations and a limited number of fungible token invariants run against pinned versions within a single development environment. This limitation is consistent with established software-engineering experiment guidance, where external validity depends on whether results from a specific experimental setting can be generalized to other systems, tools, and environments [66]. As such, the results do not generalize directly to ERC-20 extensions, Token-2022, upgradeable or bridge smart contract code, or full NFT marketplace logic. The KEVM outcome is the clearest example of this, as its failure resulted from compilation and memory issues on the WSL2 host and would not necessarily occur on a provisioned native machine; it should therefore be viewed as a usability observation within the given setting rather than as a universal assessment of the tool [24]. The scope limitation is by design, as the aim was to evaluate core invariant structures under different architectures rather than to audit all implementations. The agreement of the account validation results with independent studies such as the Neodyme pitfalls catalog provides corroborative support for the findings, though it does not establish universality [64].

Overall, the validity of this thesis depends upon the bounds within which its findings are interpreted. While these experiments have provided credible evidence to support invariant-based formal verification as a viable means of identifying and verifying important token correctness properties across both Solana and Ethereum, they have not established that any given deployed system has been completely verified. The strongest claim that can be made is that cross-platform formal verification is useful only if explicit information about the assumptions used by the verifier, the abstraction level, the degree of runtime fidelity achieved, and the boundary of what was formally proved are provided. This supports the view that validity is an ongoing matter concerning how claims are constructed and constrained, rather than something to be checked off at the end of a study [67].

7 Conclusion

The findings summarized above can now be related directly to the four research questions that guided the thesis. Each question is restated and then answered in light of the invariant framework of Chapter 4 and the verification results and analysis of Chapters 5 and 6.

7.1 Research Question 1

RQ1 asked what functional correctness properties and security invariants define the behavior of fungible ERC-20 and SPL token standards across the Ethereum and Solana ecosystems.

The thesis addressed RQ1 by creating an implementable invariant framework for defining the behavior of fungible token standards in terms of functional correctness properties and security invariants across both the Ethereum and Solana ecosystems. As established in [13], four broad families of properties that are maintained persistently throughout all valid states of a fungible token include balance consistency, supply preservation with minting constraint limitations, the validity of each transfer action, and the ability to delegate or authorize other accounts to perform actions on behalf of an account. These families were further refined into two concrete invariant sets: six ERC-20 invariants denoted ERC20-I1 through ERC20-I6, defined over the _balances, _allowances, and _totalSupply storage variables of ERC-20 compliant smart contracts [25]; and six SPL Token invariants defined over mint account and token account field values in SPL Token compliant programs [17]. The substantive answer to RQ1 is therefore represented by the validated fungible invariant framework together with an empirical observation: when an invariant appears conceptually in both ecosystems, it may impose different proof burdens depending on where the constrained state is stored, and SPL-I5, which has no conceptual analogue in ERC-20, represents the most extreme instance of this.

7.2 Research Question 2

RQ2 asked how effectively the selected formal verification tools can be used to specify and verify the identified token correctness properties.

The formal verification tools as a group were successful in describing and proving the fungible invariants; however, each tool was limited in what it could accomplish alone and therefore the tools have complementary rather than substitutable value. Certora provided the strongest deductive assurance for single-relation ERC-20 properties from source code [25].

Foundry offered the most practical developer-integrated regression testing evidence, but Foundry only guarantees the absence of counterexamples within its testing campaign. KEVM provided the strongest theoretical basis for bytecode-level assurance, yet it was not possible to use in the study environment [24]. Kani completely verified bounded Rust logic within its harness bounds. Prusti provided the most precise contract-level diagnostics [28]. MIRAI added lightweight detection of reachable runtime panics without requiring a full specification [29]. No single tool provided complete assurance across all five evaluation criteria, because each verified outcome remained conditionally dependent upon the quality of the specifications, the coverage of the harness, the level of abstraction used when modeling, and the accuracy of the verification target.

7.3 Research Question 3

RQ3 asked how architectural and execution model differences between the EVM and Solana Runtime influence the design, specification, and verification of token invariants.

Addressing RQ3, this research investigated how differences in architecture and execution models between the EVM and Solana Runtime affect the design, specification, and verification of token invariants. The findings show that these architectural and execution model differences are fundamental rather than incidental; they determine, prior to any tool being applied, where an invariant resides, what must be proved, and what structural assurances are already in place [52]. The EVM defines invariants primarily in terms of a contract's own storage, providing compactness and local checkability for arithmetic relations; however, responsibility falls to the verifier to demonstrate that no illegal write path exists within the contract [24]. EVM also places obligations such as mint authorization outside the verification artifact [25].

Conversely, Solana Runtime defines invariants over explicit accounts and their mutability, authority, and signers. This enables the runtime to verify that authority and identity have been appropriately validated and allows some properties such as transfer supply conservation to be converted into nearly structural assurances through read-only references and Rust's ownership model. This comes at the cost of imposing a new specification requirement on the programmer to correctly represent account identities and relationships. Additionally, while the runtime provides assurance regarding internal state transitions, it abstracts away a concurrent execution scheduler that sequential harnesses do not represent [8]. The Move language's resource model provides external validation of this finding, since relocating conservation

from a proof obligation into the type system eliminates that obligation entirely [70]. The practical consequence is that cross-platform invariant comparisons are only meaningful when they indicate what portion of each result was attributable to the language and account model and what portion was attributable to the proof itself.

7.4 Research Question 4

RQ4 asked about the practical implications, limitations, and trade-offs of applying formal verification across heterogeneous blockchain platforms for developers, auditors, and security engineers.

For developers, *invariants* should be treated as design artifacts with a well-defined mathematical domain from the outset. Building a *harness* is a primary task, and a successful *harness* test provides evidence that the *harness* is correctly constructed, though not necessarily that the underlying code is correct [26]. For auditors, *verified* labels must be interpreted in conjunction with the model used for verification, the *runtime* assumptions of the system, and all behaviors that were omitted [58]. For example, a proof of *SPL-I3 supply conservation* demonstrates that supply is conserved but says nothing about *SPL-I5* account validity unless that relationship was explicitly modeled [64]. For security engineers, assurance is best achieved by organizing verification into stages, each addressing a different level of correctness [57]. Each report should include an artifact specifying the *invariant* used, the tool and its version, the assumptions made, and the *runtime* features omitted from the analysis [60].

The overall trade-off in this research has been between *fidelity* and *tractability*. The tool providing the strongest deployment-level guarantee had the lowest practical usability, while those that were easiest to run produced results based on less complete models [47]. Token movements between ecosystems through bridges and interoperability layers continue to produce significant large-scale losses [71].

7.5 Future Research Directions

The scope decisions and limitations recorded in Section 6.10 open several directions for future research, each of which would extend the invariant-based, cross-platform approach beyond the boundaries established here.

The first direction addresses the target fidelity threat described in Section 6.2. The Solana tools used in this study reasoned about reconstructed versions of the MintAccount and TokenAccount models rather than directly over the deployed SPL Token program at runtime [51]. Validating the actual SPL Token program by modeling the omitted runtime characteristics, specifically account rent, cross-program invocations, program-derived addresses, and the parallel transaction scheduler, represents a direction for future work [8]. Of these omitted characteristics, the parallel transaction scheduler is the most complex, because an instruction-level harness cannot accurately reflect all possible interleaving scenarios arising from contention for a shared token account among multiple concurrently executing transactions; the complexity involved in modeling such systems is well illustrated by the specialized nature of parallel execution semantics [76]. The second direction involves completing the KEVM bytecode proof of ERC20-I3 using a provisioned native environment, thereby closing the remaining source-to-bytecode gap identified in Section 5.1.3 [24].

The second direction concerns reducing the specification effort required from developers. Since the results of each experiment depended on rules, contracts, and harnesses that the developer had to write by hand, it became apparent through this study that an incorrect specification would produce findings about the specification itself rather than the code being examined. Automatically inferring candidate invariants from execution logs may reduce this burden and potentially reveal properties that an analyst might otherwise overlook [75]. A complementary approach would take the verified invariants and apply them as runtime monitors so that non-compliant transactions can be rejected on the blockchain post-deployment, providing continuous assurance that the properties statically guaranteed in Chapter 5 are enforced in practice [60].

A further direction expands the verification scope beyond the minimum fungible token set specified above. For example, the ERC-20 extensions ERC20Burnable and ERC20Pausable, as well as the transfer fee and confidential transfer mechanisms of the Token-2022 extension, each introduce additional state and authorization requirements that would need to be verified against their own invariants [17]. Interoperability bridges represent some of the highest-value targets in the current ecosystem and continue to be among the most frequently loss-causing assets; verifying them would require invariants spanning two runtime environments simultaneously [71]. Finally, resource-oriented programming languages such as Move represent an alternative baseline for this research by encoding conservation directly into the type system; studying them could provide insight into how much of the cross-platform proof

burden is inherent to the problem versus incidental to the platform architecture [70]. The current focus of this thesis has been specifically on fungible tokens, centering on balance and supply correctness; however, non-fungible tokens present their own correctness requirements centered around unique identifiers and ownership [77]. Extending the framework to NFT standards is also warranted, particularly in relation to the Solana relationship between a metadata account and the on-chain metadata associated with a mint through the Metaplex Token Metadata Program [78], as numerous security issues related to ownership and metadata correctness have been identified across the NFT ecosystem [74]. It would also be informative to determine whether applying the same six tools to NFT standards produces the same degree of proof burden asymmetry observed for fungible tokens when shifting from balance and supply arithmetic to uniqueness and metadata correctness.

Pursued together, these directions would move invariant-based formal verification from the bounded, single-implementation case studies of this thesis toward a broader, runtime-faithful, and partially automated discipline applicable across the heterogeneous platforms and asset classes on which token value increasingly depends.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf [Accessed: May 14, 2026].

[2] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in Proc. IEEE Int. Congr. Big Data (BigData Congress), Honolulu, HI, USA, 2017, pp. 557–564, doi: 10.1109/BigDataCongress.2017.85.

[3] V. Buterin, "A next-generation smart contract and decentralized application platform," Ethereum White Paper, 2014. [Online]. Available: https://ethereum.org/en/whitepaper/ [Accessed: Mar. 22, 2026].

[4] G. Wood, "Ethereum: A secure decentralized generalized transaction ledger," Ethereum Yellow Paper, 2014.

[5] A. Yakovenko, "Solana: A new architecture for a high-performance blockchain," Solana White Paper, 2018. [Online]. Available: https://solana.com/solana-whitepaper.pdf [Accessed: Apr. 3, 2026].

[6] M. Loporchio, D. Di Francesco Maesa, A. Bernasconi, and L. Ricci, "Comparing Ethereum fungible and non-fungible tokens: An analysis of transfer networks," Applied Network Science, vol. 9, Art. no. 72, 2024, doi: 10.1007/s41109-024-00682-8.

[7] K. Bhargavan et al., "Formal verification of smart contracts," in Proc. ACM Workshop Programming Languages and Analysis for Security (PLAS), 2016.

[8] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, and K. Pohl, "Fuzz on the Beach: Fuzzing Solana smart contracts," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Copenhagen, Denmark, 2023, pp. 1197–1211, doi: 10.1145/3576915.3623178.

[9] M. Conti, S. Kumar, C. Lal, and S. Ruj, "A survey on security and privacy issues of Bitcoin," IEEE Communications Surveys & Tutorials, vol. 20, no. 4, pp. 3416–3452, 2018.

[10] K. Wüst and A. Gervais, "Do you need a blockchain?" in Proc. 1st Crypto Valley Conf. Blockchain Technology, Zug, Switzerland, 2018, pp. 45–54.

[11] Ethereum Foundation, "Introduction to smart contracts," Ethereum Developer Documentation. [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/ [Accessed: Mar. 30, 2026].

[12] Ethereum Foundation, "Ethereum whitepaper," Ethereum.org. [Online]. Available: https://ethereum.org/en/whitepaper/ [Accessed: May 2, 2026].

[13] F. Vogelsteller and V. Buterin, "ERC-20: Token standard," Ethereum Improvement Proposals, EIP-20, 2015. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20 [Accessed: Apr. 18, 2026].

[14] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), London, United Kingdom, 2019, pp. 1503–1520, doi: 10.1145/3319535.3345664.

[15] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts," in Principles of Security and Trust, 2017, pp. 164–186.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Vienna, Austria, 2016, pp. 254–269, doi: 10.1145/2976749.2978309.

[17] Solana Foundation, "Accounts," Solana Documentation. [Online]. Available: https://solana.com/docs/core/accounts [Accessed: Mar. 25, 2026].

[18] Solana Foundation, "Transactions," Solana Documentation. [Online]. Available: https://solana.com/docs/core/transactions [Accessed: Apr. 27, 2026].

[19] Anza, "Solana runtime on a Solana validator," Anza Documentation. [Online]. Available: https://docs.anza.xyz/validator/runtime [Accessed: May 21, 2026].

[20] R. Ji, N. He, L. Wu, H. Wang, G. Bai, and Y. Guo, "DEPOSafe: Demystifying the fake deposit vulnerability in Ethereum smart contracts," in Proc. 25th Int. Conf. Engineering of Complex Computer Systems (ICECCS), 2020, pp. 125–134, doi: 10.1109/ICECCS51672.2020.00021.

[21] Solana Foundation, "Tokens on Solana," Solana Documentation. [Online]. Available: https://solana.com/docs/tokens [Accessed: Apr. 9, 2026].

[22] Solana Foundation, "Security best practices," Anchor Documentation. [Online]. Available: https://www.anchor-lang.com/docs/references/security-exploits [Accessed: Jun. 1, 2026].

[23] S. Somin, Y. Altshuler, and A. Pentland, "Crypto-asset trading on top of Ethereum Blockchain comprehensive dataset," Scientific Data, vol. 12, Art. no. 1407, 2025, doi: 10.1038/s41597-025-05662-w.

[24] E. Hildenbrandt et al., "KEVM: A complete formal semantics of the Ethereum Virtual Machine," in Proc. IEEE Computer Security Foundations Symp. (CSF), 2018, pp. 204–217.

[25] Certora, "Certora Verification Language: Specification files," Certora Documentation. [Online]. Available: https://docs.certora.com/en/latest/docs/cvl/overview.html [Accessed: Apr. 14, 2026].

[26] Foundry, "Invariant testing," Foundry Documentation. [Online]. Available: https://getfoundry.sh/forge/advanced-testing/invariant-testing/ [Accessed: Mar. 28, 2026].

[27] Kani, "Getting started: The Kani Rust verifier," Kani Documentation. [Online]. Available: https://model-checking.github.io/kani/ [Accessed: May 8, 2026].

[28] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The Prusti project: Formal verification for Rust," in NASA Formal Methods, 2022, pp. 88–108.

[29] Facebook Experimental, "MIRAI: Rust mid-level IR abstract interpreter," GitHub Repository. [Online]. Available: https://github.com/facebookexperimental/MIRAI [Accessed: Apr. 30, 2026].

[30] N. Ivanov, H. Guo, and Q. Yan, "Rectifying administrated ERC20 tokens," in Information and Communications Security, ICICS 2021, Lecture Notes in Computer Science, vol. 12918, Springer, 2021, pp. 22–37, doi: 10.1007/978-3-030-86890-1_2.

[31] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in Proc. 30th USENIX Security Symp., San Francisco, CA, USA, 2021, pp. 1325–1341.

[32] S. Cui, G. Zhao, Y. Gao, T. Tavu, and J. Huang, "VRust: Automated vulnerability detection for Solana smart contracts," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Los Angeles, CA, USA, 2022, pp. 639–652, doi: 10.1145/3548606.3560552.

[33] J. Navas, "Formal verification of SPL Token 2022," Certora Blog, Aug. 2023. [Online]. Available: https://medium.com/certora/formal-verification-of-spl-token-2022-d521db465c2e [Accessed: May 19, 2026].

[34] Solana Foundation, "Program derived addresses," Solana Documentation. [Online]. Available: https://solana.com/docs/core/pda [Accessed: Jun. 6, 2026].

[35] Solana Foundation, "Tokens on Solana," Solana Documentation. [Online]. Available: https://solana.com/docs/tokens [Accessed: Apr. 22, 2026].

[36] Certora, "Reviewing token extensions on Solana using formal verification," Certora Blog, 2024. [Online]. Available: https://www.certora.com/blog/token-extensions-audit [Accessed: May 26, 2026].

[37] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for Solidity smart contracts," in Proc. VSTTE, 2019.

[38] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "VerX: Safety verification of smart contracts," in Proc. IEEE Symp. Security and Privacy (SP), 2020.

[39] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), 2018.

[40] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Toward verifying Ethereum smart contract bytecode in Isabelle/HOL," in Proc. ACM SIGPLAN Int. Conf. Certified Programs and Proofs, 2018.

[41] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2018.

[42] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in Proc. ACM Joint European Software Engineering Conf. and Symp. Foundations of Software Engineering (ESEC/FSE), 2020.

[43] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for Solidity smart contracts," in Proc. ACM/IEEE Int. Conf. Software Engineering (ICSE), 2020.

[44] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in Verification, Model Checking, and Abstract Interpretation, LNCS, vol. 9583, Springer, 2016, pp. 41–62.

[45] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in Proc. TACAS, 2004.

[46] Y. Matsushita, T. Tsukada, and N. Kobayashi, "RustHorn: CHC-based verification for Rust programs," ACM Transactions on Programming Languages and Systems, vol. 43, no. 4, Art. no. 15, pp. 1–54, 2021, doi: 10.1145/3462205.

[47] H. Garavel, M. H. ter Beek, and J. van de Pol, "The 2020 expert survey on formal methods," in Proc. 25th Int. Conf. Formal Methods for Industrial Critical Systems (FMICS), LNCS, vol. 12327, Springer, 2020, pp. 3–69.

[48] Solidity Documentation, "Layout of state variables in storage and transient storage," Solidity Docs. [Online]. Available: https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html [Accessed: Mar. 20, 2026].

[49] Ethereum Foundation, "Ethereum execution layer specifications," GitHub Repository. [Online]. Available: https://github.com/ethereum/execution-specs [Accessed: May 11, 2026].

[50] Solana Foundation, "Program derived addresses (PDAs)," Solana Documentation. [Online]. Available: https://solana.com/docs/core/pda [Accessed: Jun. 9, 2026].

[51] T. Cloosters, P. Winkler, J.-R. Giesen, G. Karame, and L. Davi, "SseRex: Practical symbolic execution of Solana smart contracts," arXiv:2603.16349, 2026.

[52] F. B. Schneider, "Enforceable security policies," ACM Transactions on Information and System Security, vol. 3, no. 1, pp. 30–50, 2000.

[53] S. Blackshear et al., "Move: A language with programmable resources," Libra Association, 2019.

[54] C. A. R. Hoare, "An axiomatic basis for computer programming," Communications of the ACM, vol. 12, no. 10, pp. 576–580, 1969.

[55] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," in Proc. ACM SIGPLAN Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2018.

[56] A. Borgida, J. Mylopoulos, and R. Reiter, "On the frame problem in procedure specifications," IEEE Transactions on Software Engineering, vol. 21, no. 10, pp. 785–798, 1995.

[57] S. Chaliasos, M. A. Charalambous, L. Zhou, R. Galanopoulou, A. Gervais, D. Mitropoulos, and B. Livshits, "Smart contract and DeFi security tools: Do they meet the needs of practitioners?" in Proc. IEEE/ACM 46th Int. Conf. Software Engineering (ICSE), Lisbon, Portugal, 2024, pp. 716–728.

[58] D. Feng, R. Hitsch, K. Qin, A. Gervais, R. Wattenhofer, Y. Yao, and Y. Wang, "DeFi auditing: Mechanisms, effectiveness, and user perceptions," in Financial Cryptography and Data Security, FC 2023 International Workshops, LNCS, vol. 13953, Springer, Cham, 2024, pp. 320–336.

[59] G. Qiao and P. P. Paul, "Human side of smart contract fuzzing: An empirical study," arXiv:2506.07389, 2025.

[60] Z. Chen, Y. Liu, S. M. Beillahi, Y. Li, and F. Long, "Demystifying invariant effectiveness for securing smart contracts," Proceedings of the ACM on Software Engineering, vol. 1, no. FSE, Art. no. 79, 2024.

[61] A. Groce, J. Feist, G. Grieco, and M. Colburn, "What are the actual flaws in important smart contracts and how can we find them?" in Financial Cryptography and Data Security, FC 2020, LNCS, vol. 12059, Springer, Cham, 2020, pp. 634–653.

[62] OWASP, "Smart Contract Security Verification Standard (SCSVS)," OWASP Smart Contract Security Project. [Online]. Available: https://scs.owasp.org/SCSVS/ [Accessed: Apr. 6, 2026].

[63] Enterprise Ethereum Alliance, "EEA EthTrust Security Levels Specification Version 3," Enterprise Ethereum Alliance, Inc., 2025. [Online]. Available: https://entethalliance.org/specs/ethtrust-sl/ [Accessed: May 30, 2026].

[64] Neodyme Audit Team, "Solana smart contracts: Common pitfalls and how to avoid them," Neodyme Blog, 2021. [Online]. Available: https://neodyme.io/en/blog/solana_common_pitfalls/ [Accessed: Jun. 12, 2026].

[65] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," Empirical Software Engineering, vol. 14, no. 2, pp. 131–164, 2009.

[66] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in Software Engineering. Berlin, Heidelberg: Springer, 2012.

[67] D. I. K. Sjøberg and G. R. Bergersen, "Construct validity in software engineering," IEEE Transactions on Software Engineering, vol. 49, no. 3, pp. 1374–1396, 2023.

[68] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," ACM Computing Surveys, vol. 54, no. 7, Art. no. 148, pp. 1–38, 2021.

[69] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "SoK: Decentralized finance (DeFi) attacks," in Proc. IEEE Symp. Security and Privacy (SP), San Francisco, CA, USA, 2023, pp. 2444–2461.

[70] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, and D. L. Dill, "The Move Prover," in Computer Aided Verification (CAV 2020), LCS, vol. 12224, Springer, Cham, 2020, pp. 137–150.

[71] N. Belenkov, V. Callens, A. Murashkin, M. Derka, J. Gorzny, and S. Bayhan, "SoK: A review of cross-chain bridge hacks in 2023," arXiv:2501.03423, 2025.

[72] M. di Angelo and G. Salzer, "A survey of tools for analyzing Ethereum smart contracts," in Proc. IEEE Int. Conf. Decentralized Applications and Infrastructures (DAPPCON), Newark, CA, USA, 2019, pp. 69–78.

[73] M. Wöhrer and U. Zdun, "Smart contracts: Security patterns in the Ethereum ecosystem and Solidity," in Proc. IEEE Int. Workshop Blockchain Oriented Software Engineering (IWBOSE), Campobasso, Italy, 2018, pp. 2–8.

[74] D. Das, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, "Understanding security issues in the NFT ecosystem," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), Los Angeles, CA, USA, 2022, pp. 667–683.

[75] Y. Liu and Y. Li, "InvCon: A dynamic invariant detector for Ethereum smart contracts," in Proc. 37th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), Rochester, MI, USA, 2022, Art. no. 160.

[76] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing," in Proc. 28th ACM SIGPLAN Annu. Symp. Principles and Practice of Parallel Programming (PPoPP), Montreal, QC, Canada, 2023, pp. 232–244.

[77] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "ERC-721: Non-fungible token standard," Ethereum Improvement Proposals, EIP-721, 2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721 [Accessed: May 5, 2026].

[78] Metaplex Foundation, "Token Metadata," Metaplex Documentation. [Online]. Available: https://www.metaplex.com/docs/smart-contracts/token-metadata [Accessed: Jun. 15, 2026].