Journal of Software Engineering Research and Development, 2025, 13:17, doi: 10.5753/jserd.2025.5061

Comparative Analysis of Hoare Logic-Based Formal Verification Tools for Solidity Smart Contracts

Alexandre Mota [**Centro de Informática - Universidade Federal de Pernambuco** | ***acm@cin.ufpe.br***]
Manoel Villarim [**Centro de Informática - Universidade Federal de Pernambuco** | ***mvaf@cin.ufpe.br***]
Juliano Iyoda [**Centro de Informática - Universidade Federal de Pernambuco** | ***jmi@cin.ufpe.br***]
Márcio Cornélio [**Centro de Informática - Universidade Federal de Pernambuco** | ***mlc2@cin.ufpe.br***]

Abstract Formal verification of smart contracts is widely regarded as an effective method for ensuring correctness and security properties across all possible executions. Its practical relevance has been driven by the availability of automatic verification tools that discharge intricate proofs. Another area of growing interest is the integration of specification paradigms — for example, combining Hoare-logic–style specifications (pre/postconditions and invariants) with SMT and symbolic reasoning — so that each technique can precisely capture complementary aspects of contract behavior. In this article we present a comparative analysis[1] of four leading Solidity verification tools — solc-verify, SMTChecker, VeriSmart and the Certora Prover — and define what is meant here by a formal verification tool: a system that provides mathematically rigorous proofs that stated properties hold for every possible execution of a contract. We also describe a consistent evaluation framework that considers the Solidity version support, the preservation of the original contract structure, the local execution capability, the verification time, and the modeling-language requirements, among other criteria. We used the ERC-20 token standard as a benchmark and applied this framework to obtain empirical evidence of each tool's capabilities and limitations. Our results expose substantial variability in the tools performances that undermines their trustworthiness in practice and highlights a gap between an academic tool capabilities and the industrial requirements. Finally, we discuss how these findings can inform developers and researchers in selecting appropriate verification tools, thereby contributing to improved smart contract security and reliability.

Keywords: *Smart Contracts, Solidity, solc-verify, SMTChecker, VeriSmart, Certora*

1 Introduction

The blockchain is a distributed and immutable ledger made up of ever-expanding lists of records, or blocks, that are safely connected by cryptographic hashes. Ethereum (Wood, 2014) is a decentralized blockchain with smart contract capabilities. According to the rules of a contract or agreement, a smart contract is a computer program or transaction protocol designed to automatically carry out, regulate, or record events and actions (Szabo, 1997). This technological stack collectively forms the foundation for decentralized applications and programmable financial instruments that operate without traditional intermediaries (Ante, 2020).

Solidity (Soliditylang.org, 2024b) is a high-level programming language specifically designed for writing Ethereum smart contracts. Solidity is employed in diverse application domains such as decentralized finance (DeFi), tokenization protocols, supply chain verification systems, and distributed governance mechanisms. Within this ecosystem, the ERC20 token standard (Ethereum.org, 2024) has emerged as the predominant interface specification for implementing fungible tokens on the Ethereum network. It provides a standardized framework that ensures interoperability and consistent functionality across different token implementations.

The capacity of smart contracts to facilitate complex multi-party transactions without requiring trust between participants has fundamentally reconfigured the architecture of digital agreement systems. The inherent immutability of the blockchain technology makes a smart contract code unalterable after deployment. This feature prevents the possibility of patching bugs or security vulnerabilities after deployment and thereby establishes a risk of significant, irrecoverable financial losses. Therefore it is critical for a software engineer to adopt a robust formal verification tool in order to ensure the correctness and the security of smart contracts before deployment (Kulik et al., 2021).

While several formal verification tools for Solidity smart contracts have been developed, there is a lack of comprehensive comparative analysis to guide developers and researchers in selecting the most appropriate tool for their needs. This article aims to address this gap by providing an in-depth evaluation of four prominent Solidity formal verification tools: solc-verify (Hajdu and Jovanović, 2019), SMTChecker (Alt and Reitwiessner, 2018a), VeriSmart (So et al., 2020), and the Certora Prover (Certora.com, 2024).

Our research objective is to systematically evaluate these tools applicability for developers and security auditors. We assess them across several practical criteria that directly impact tool adoption and effectiveness, including: (i) compatibility with recent Solidity versions; (ii) ability to verify contracts with minimal adaptations to the original source code; (iii) support for local execution without reliance on external services; (iv) efficiency of the verification time; (v) requirement for adopting a domain-specific language for property

specification; and (vi) expressiveness in capturing complex security properties. These criteria were selected based on their significance for streamlining the verification workflow, thus minimizing the learning curve for developers, and ensuring the verification process can be integrated into existing development pipelines. We have chosen to apply the tools to the widely-used ERC20 token standard (Ethereum.org, 2024). ERC20 serves as an ideal benchmark due to its ubiquity in the blockchain ecosystem, its standardized implementation across platforms, and its direct relevance to real-world financial applications. Therefore, the adoption of ERC20 for our analysis will offer insights into the verification tools unique strengths, limitations, and trade-offs in a real industrial setting.

Our evaluation comes from the application of metrics that assess the practical challenges of integrating these tools into a development workflow, which is a recognized gap in the literature. It is crucial to note that, at this stage, our study does not measure bug-finding effectiveness, but the practical feasibility to use these tools. For research focused specifically on vulnerability-detection capabilities, please refer to the works of Almakhour et al. (2020), Garfatta et al. (2021), Happersberger et al. (2023), Wei et al. (2023), and Bartoletti et al. (2024).

The results of our analysis reveal significant variations in the capabilities and applicability of these tools. We find that, while some tools excel in specific areas, no single tool provides dominance across all criteria. Our results highlight the need for developers to carefully consider the tools specific capabilities and limitations before selecting one.

This article presents the following contributions.

- A systematic comparison of four leading Solidity formal verification tools across multiple practical criteria;
- A detailed analysis on how each tool handles the formalization and the verification of the ERC20 token standard;
- Insights into the real-world practical applicability of these tools, including limitations when dealing with complex, real smart contracts;
- Identification of gaps in the current tool capabilities, thus providing guidance for future research and development in smart contract verification.

By providing this comparative analysis, our work aims to empower Ethereum developers and security experts to make informed decisions when selecting formal verification tools, thus ultimately contributing to the development of more secure and reliable smart contracts. The findings also highlight areas for improvement in existing tools and suggest directions for future research in smart contract verification.

The structure of this article is as follows. The article provides an overview of the Solidity programming language through the ERC20 token in Section 2. In Section 3, we present the Solidity formal verification tools under investigation as well as the Hoare Logic assertions to be verified. The findings of this evaluation are discussed in Section 4, revealing the unique strengths, weaknesses, and trade-offs of each tool. The article then reviews related work in Section 5 and concludes in Section 6.

2 Solidity and the ERC20 Token

Solidity (Soliditylang.org, 2024b) is an object-oriented programming language designed for writing smart contracts. It is widely supported by various blockchain platforms. The initial release, version 0.1.2, was made available in August 2015, and since then, Solidity has been under continuous development with sponsorship from the Ethereum Foundation. At the time of writing, the current version of Solidity is 0.8.31.

Although we present here the verification of the function `transfer()` for conciseness, our work takes into account the entire contract with all its functions. Listing 1 is an excerpt of a standard implementation of the ERC20 token interface[2] written in Solidity. Listing 1 provides a set of functions and events that allow for the creation, transfer, and management of ERC20 tokens.

Here is a breakdown of the key components of the code:

- License (line 1): The SPDX-License serves as a standardized format for documenting and communicating software license information within the Ethereum ecosystem. In this case, the license used was the MIT license;
- Solidity version (line 2): The statement `pragma solidity ^0.8.20` declares that this code may be compiled with version 0.8.20 or higher;
- Imports (lines 3–6): The code imports several interfaces and utility contracts, including IERC20, IERC20Metadata, Context, and IERC20Errors. These libraries provide standard definitions and functionality for ERC20 tokens;
- Contract definition (line 8): The ERC20 contract is defined as an abstract contract. This means that it cannot be directly deployed as a contract but must be inherited by other contracts;
- Inheritance (line 8): The keyword `is` states that the ERC20 contract inherits the properties of the contract `Context` and implements the interfaces `IERC20` and `IERC20Metadata`;
- State variables (lines 9–13): The contract maintains several state variables. The mapping `_balances` stores a key-value data to track the token balances per address, while the mapping `_allowances` stores the approved allowances to spend tokens on behalf of another address. The variable `_totalSupply` stores the total supply of tokens, `_name` stores the name of the token, and `_symbol` stores the symbol of the token;
- Constructor (line 15): The constructor initializes the name and the symbol of the token;
- Public function `transfer()` (lines 24–28): The `transfer()` function allows a user to transfer tokens to another address. It checks that the recipient address is not the zero address and that the sender has a sufficient balance, by calling the internal function `_transfer()`;
- Internal functions (lines 30–57): The `_transfer()` function is an internal function that performs the actual token transfer. It checks that the sender and the recipient addresses are not the zero address (lines 31–35) and updates the balances accordingly (line 37). The `_update()` function is an internal function that updates the token balances and total supply when a transfer occurs. It ensures that the total supply is updated correctly and that the sender has a sufficient balance (line 45);
- Events (line 55): The statement `emit Transfer(from, to, value)` sends the event `Transfer(from, to, value)` to

[2] Particularly, this implementation comes from OpenZeppelin. `https://www.openzeppelin.com`

the log system which can be further analyzed by the developers.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
import {IERC20} from "./IERC20.sol";
import {IERC20Metadata} from "./extensions/
    IERC20Metadata.sol";
import {Context} from "../../utils/Context.sol";
import {IERC20Errors} from "../../interfaces/
    draft-IERC6093.sol";

abstract contract ERC20 is Context, IERC20,
    IERC20Metadata, IERC20Errors {
    mapping(address account => uint256) private
        _balances;
    mapping(address account => mapping(address
        spender => uint256)) private _allowances;
    uint256 private _totalSupply;
    string private _name;
    string private _symbol;

    constructor(string memory name_, string
        memory symbol_) {
        _name = name_; _symbol = symbol_;
    }
    ... // Previous functions before transfer
    /
     * Requirements:
     * - `to` cannot be the zero address.
     * - the caller must have a balance of at
         least `value`.
     */
    function transfer(address to, uint256 value)
        public virtual returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, value);
        return true;
    }

    function _transfer(address from, address to,
        uint256 value) internal {
        if (from == address(0)) {
            revert ERC20InvalidSender(address(0))
                ;
        }
        if (to == address(0)) {
            revert ERC20InvalidReceiver(address
                (0));
        }
        _update(from, to, value);
    }

    function _update(address from, address to,
        uint256 value) internal virtual {
        if (from == address(0)) {
            _totalSupply += value;
        } else {
            uint256 fromBalance = _balances[from
                ];
            if (fromBalance < value) {
                revert ERC20InsufficientBalance(
                    from, fromBalance, value);
            }
            unchecked { _balances[from] =
                fromBalance - value; }
        }
        if (to == address(0)) {
            unchecked { _totalSupply -= value; }
        } else {
            unchecked { _balances[to] += value; }
        }
        emit Transfer(from, to, value);
    }
    ... // Rest of the smart contract functions
}
```

Listing 1: OpenZeppelin's ERC20 Token.

2.1 Potential Errors in the ERC20 contract

Our four analyzed tools could find vulnerabilities related to the `transfer()` function, each with distinct verification approaches and overlapping error detection capabilities.

All tools identify standard vulnerabilities:

- Arithmetic over/underflows and division by zero
- Array bounds violations (empty pop, out-of-bounds access)
- Control flow issues (unreachable code, trivial conditions)
- Transfer-related problems (insufficient funds)

VeriSmart extends detection to include **Suicidal** and **Ether-Leaking** vulnerabilities, while Certora's CVL enables custom rule verification beyond standard Solidity constructs.

However, it is important to note that the evolution of the Solidity language has made the compiler capable of automatically handling many of those vulnerabilities through a built-in safety mechanism that revert transactions and preserve the blockchain state integrity when such conditions are detected. As a result, the verification properties of primary interest to programmers have shifted toward those based on Hoare logic principles, specifically focusing on the specification and the verification of preconditions, postconditions, and loop invariants.

3 The verification tools

This section presents an overview and an analysis of four prominent formal verification tools for Solidity smart contracts: solc-verify (Hajdu and Jovanović, 2019), SMTChecker (Alt and Reitwiessner, 2018a), VeriSmart (So et al., 2020), and the Certora Prover (Certora.com, 2024). These tools are tied to a Hoare logic-based approach: that is, they focus on formal verification in terms of preconditions, postconditions, and invariants, which is why we selected them for our investigation. These tools represent a range of approaches to formal verification, from those integrated directly into the Solidity compiler to standalone solutions employing specialized verification languages. We examine each key feature, methodology, and underlying technology for formalizing and verifying smart contract properties.

This section presents an overview and an analysis of four prominent formal verification tools for Solidity smart contracts: solc-verify (Hajdu and Jovanović, 2019), SMTChecker (Alt and Reitwiessner, 2018a), VeriSmart (So et al., 2020), and the Certora Prover (Certora.com, 2024) (see Tables 1 and 2 for a summarized comparison). These tools represent a range of approaches to formal verification, from those integrated directly into the Solidity compiler to standalone solutions employing specialized verification languages. Our selection highlights the diversity in

Table 1. Comparative Analysis of Formal Verification Tools (Part 1)

Tool	Underlying Technology	Specification Language	Integration
solc-verify	Boogie IVL, SMT Solvers	Comment-based DSL	Standalone tool
SMTChecker	SMT Solvers (e.g., Z3)	Solidity's `require`/`assert`	Built into Solidity Compiler
VeriSmart	Symbolic Execution, SMT	Solidity's `require`/`assert`	Standalone tool
Certora Prover	EVM-level analysis, SMT	Certora Verification Language (CVL)	Standalone tool

Table 2. Comparative Analysis of Formal Verification Tools (Part 2)

Tool	Solidity Version Support	Execution Environment	Analysis Level	Open Source
solc-verify	Older versions (e.g., 0.5.11)	Local	Solidity Source Code	Yes
SMTChecker	Supports recent versions	Local	Solidity Source Code	Yes
VeriSmart	Older versions (e.g., 0.5.11)	Local	Solidity Source Code	Yes
Certora Prover	Supports recent versions	Cloud-based/Local	EVM Bytecode	No (Free for non-commercial use)

verification approaches and adoption: SMTChecker represents a compiler-integrated academic tool, solc-verify is the most Hoare-logic adherent tool, VeriSmart combines symbolic execution with SMT solving, and the Certora Prover is an industry-standard EVM-level tool. Together, they showcase the spectrum of verification approaches (Hoare logic, SMT solving, symbolic execution) and different levels of industry and academic adoption. We examined, for each tool, its key features, its underlying technologies, and its methodology for formalizing and verifying smart contract properties.

In order to evaluate the capabilities of the various Solidity formal verification tools, it is also necessary to understand how each tool formalizes and represents the core functionality of the ERC20 token contract (focusing on the `transfer()` function for conciseness). This section examines the approaches taken by SMTChecker, VeriSmart, solc-verify, and the Certora Prover in formalizing the ERC20 contract. The differences in their formalization strategies provide insights into the strengths and limitations of each tool when it comes to verifying the correctness and security properties of this commonly used smart contract standard.

The basic workflow in the following sections is: the developer runs `<command>` (sometimes by using an IDE), the corresponding tool analyzes the given Solidity files, and outputs a proof or a counterexample. Tools requiring DSL (like solc□verify and Certora) mandate writing `@notice` or CVL specs, whereas SMTChecker and VeriSmart use native `require`/`assert`.

3.1 solc-verify

The solc-verify tool (Hajdu and Jovanović, 2019) provides a significant contribution to the formal verification of Solidity smart contracts. This open-source solution leverages the power of the Boogie intermediate verification language (Le Goues et al., 2011) in order to provide a robust and efficient mechanism for automatically verifying safety properties of smart contracts.

At its core, solc-verify employs a two-step translation process. First, it converts a Solidity source code into Boogie, an intermediate verification language developed by Microsoft Research. Boogie serves as a bridge between the high-level semantics of Solidity and the low-level logical formulae required by automated theorem provers. This intermediate representation captures the essential semantics of the original Solidity code while abstracting away unnecessary details, thus facilitating a more efficient verification.

In the second step, the Boogie representation is fed into automated theorem provers, typically Satisfiability Modulo Theories (SMT) solvers such as Z3 (Moura and Bjørner, 2008). These solvers attempt to prove or disprove the specified safety properties, providing formal guarantees about the contract behavior under various conditions.

One of the key strengths of solc-verify is its comprehensive coverage of common smart contract vulnerabilities. It can detect and prove the absence of issues such as: integer overflows and underflows, division by zero, array out-of-bounds access, reentrancy vulnerabilities, assertion violations, and unchecked external calls.

Moreover, solc-verify allows developers to specify customized invariants and post-conditions, thus enabling the verification of application-specific properties that go beyond the standard general safety checks.

A distinguishing feature of solc-verify is its seamless integration with the Solidity compiler toolchain. By operating as a compiler plugin, it can be invoked directly during the compilation process, making formal verification an integral part of the development workflow. This tight integration reduces the barriers to adopt solc-verify and encourages developers to incorporate formal methods into their daily practices.

The tool modular architecture allows for extensibility, enabling researchers and developers to add support for new Solidity features or to implement additional verification techniques. This flexibility has contributed to its adoption in both academic research and industry applications.

Performance-wise, solc-verify employs several optimization techniques to manage the complexity of verification tasks. These include modular verification, where contracts are analyzed component-wise, and abstraction refinement, which iteratively refines the precision of the analysis based on counterexamples.

It is worth noting that while solc-verify is highly effective for many types of smart contracts, it may face limitations when dealing with extremely complex contracts or those involving intricate mathematical operations. In such cases, it may require manual intervention to refactor the contract or the usage of complementary verification techniques.

3.1.1 ERC20 formalization for solc-verify

The solc-verify tool uses a comment-based Domain-Specific Language (DSL) in order to specify the preconditions and the postconditions as depicted in the Listing 2.

Each occurrence of the keyword `@notice` defines a solc-verify specification. In our case, the first occurrence is the contract invariant (line 2), which is defined outside the contract and uses the keyword `invariant` to indicate its purpose. Specifically, with the assistance of the internal function `__verifier_sum_uint()`, the invariant specifies that the sum of all balances equals the total supply of the contract. The postcondition asserts a successful execution of the `transfer()` function, where the conditions `msg.sender != address(0)`, `to != address(0)`, and `__verifier_old_uint(_balances[msg.sender]) >= value` must be satisfied (line 8). This postcondition checks that the balances are updated as expected, i.e. the sender balance decreases, and the recipient balance increases by the same amount (`value`). The other disjunctive cases (line 10) correspond to exceptional situations where the function reverts. If such cases occur, the state remains unchanged.

```
1 /
2 * @notice invariant __verifier_sum_uint(_balances
      ) == _totalSupply
3 */
4 contract ERC20 is Context, IERC20, IERC20Metadata
      , IERC20Errors {
5     ... // Previous elements
6     /
7     * @notice postcondition
8     * (msg.sender != address(0) && to != address
          (0) && __verifier_old_uint(_balances[msg.
          sender]) >= value && _balances[msg.sender
          ] == __verifier_old_uint(_balances[msg.
          sender]) - value && _balances[to] ==
          __verifier_old_uint(_balances[to]) +
          value)
9     * ||
10    * ((msg.sender == address(0) || to == address
          (0) || __verifier_old_uint(_balances[msg.
          sender]) < value) && _balances[msg.sender
          ] == __verifier_old_uint(_balances[msg.
          sender]) && _balances[to] ==
          __verifier_old_uint(_balances[to]) )
11    */
12    function transfer(address to, uint256 value)
          public returns (bool) {
13        address owner = _msgSender();
14
15        _transfer(owner, to, value);
16
17        return true;
18    }
19    ... // Rest of the smart contract functions
20 }
```

Listing 2: solc-verify specification.

3.2 SMTChecker and VeriSmart

This section examines SMTChecker and VeriSmart, two formal verification tools for Ethereum smart contracts that share similarities in their approach to contract formalization. Both tools leverage on Satisfiability Modulo Theories (SMT), albeit each with distinct implementations and capabilities.

SMTChecker (Alt and Reitwiessner, 2018b), developed and maintained by the Ethereum Foundation (Solidity-lang.org, 2024a), is the standard formal verification tool in the Ethereum ecosystem. It employs SMT solvers (de Moura and Bjørner, 2009) to automatically prove the correctness of Solidity contracts with respect to a predefined set of properties. The tool's primary functionality is to detect and prevent common vulnerabilities such as integer overflows, division by zero operations, and assertion violations.

At its core, SMTChecker operates by encoding the semantics of Solidity contracts into Constrained Horn Clauses (CHCs) (Fedyukovich et al., 2018). CHCs provide a powerful and flexible formalism for representing the behavior of smart contracts, thus capturing complex safety and liveness properties. These clauses are then passed to state-of-the-art SMT solvers, which attempt to either find satisfying assignments or to prove unsatisfiability, thereby verifying or refuting the specification.

SMTChecker supports two main analysis modes: CHC and Bounded Model Checking (BMC). While this article focuses on the CHC mode (because it is exhaustive), it is worth noting that the BMC mode allows for bounded verification of contract properties. However, such verification can detect

bugs that occur within a particular number of transaction executions (i.e. it is not exhaustive despite being more feasible in practice). Regardless of these two modes, SMTChecker is directly integrated into the Solidity compiler. This allows developers to perform formal verification as part of their regular compilation process. This integration significantly lowers the barrier to adopt formal methods in smart contract development.

VeriSmart (So et al., 2020), on the other hand, is an automated formal verification tool that combines symbolic execution (Cadar et al., 2011) with SMT providing a comprehensive analysis of Solidity contracts. While SMTChecker focuses primarily on safety properties, VeriSmart extends its scope to include functional correctness, security vulnerabilities, and even gas consumption analysis.

The symbolic execution engine in VeriSmart systematically explores the state space of a smart contract, generating path conditions that represent execution traces. These path conditions are then translated into SMT formulas and are passed to an SMT solver for analysis. This approach allows VeriSmart to reason about complex contract behaviors and inter-contract interactions that may be challenging to capture using CHCs alone.

One of VeriSmart distinguishing features is its ability to generate detailed, actionable reports. When issues are detected, the tool provides not only the location and nature of the problem but also suggested fixes. This feature significantly enhances its utility in practical smart contract development and auditing scenarios.

SMTChecker supports the latest Solidity language version (contrary to VeriSmart), and both SMTChecker and VeriSmart can handle complex contract structures, including inheritance and library usage. However, they differ in their handling of certain Solidity-specific concepts. For instance, SMTChecker has built-in models for Ethereum global variables and common cryptography functions, while VeriSmart may require additional annotations or abstractions for such elements.

It is important to note that while both tools aim for soundness (i.e. no false negatives), they may produce false positives due to over-approximations or limitations in their underlying solvers. Additionally, both tools face challenges when dealing with contracts that involve complex mathematical operations or rely heavily on external calls, as these aspects can lead to state space explosion or undecidability in the verification process.

3.2.1 ERC20 formalization for SMTChecker and VeriSmart

Both SMTChecker and VeriSmart use the Solidity language itself as a basis to obtain preconditions (from `requires`) and postconditions (from `asserts`) as can be seen in Listing 3.

Listing 3 is a simplified version of the Solidity code in Listing 1. The main elements to be noticed are the annotated preconditions (`requires`) and postconditions (`asserts`) as explained below.

- Precondition as documented originally: The `require(msg.sender != address(0))` and `require(to != address(0))` statements ensure that the sender and the recipient addresses are not the zero address (0x0). The `require(_balanceOwner >= value)` statement checks that the sender has a sufficient balance to perform the transfer;
- Additional preconditions (arithmetic checks): The precondition `require(value <= type(uint256).max - _balanceTo && _balanceTo` $\leq$ `type(uint256).max - value)` statement checks that the transfer value does not cause the recipient balance to overflow the maximum value of a `uint256` data type. As VeriSmart does not support the current Solidity version (at the time of writing, it is version 0.8.31), it does not accept `type(uint256).max`. Therefore, we had to type the maximum unsigned integer itself (we even tried the expression `2256-1`, but with no success);
- Assert Statements: The `assert(_balances[owner] == _balanceOwner - value)` and `assert(_balances[to] == _balanceTo + value)` statements ensure that the balances of the sender and the recipient are updated correctly after the transfer.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
3 contract ERC20 {
4     ... // Previous elements
5     function transfer(address to, uint256 value)
          public returns (bool) {
6         address owner = _msgSender();
7         uint256 _balanceOwner = _balances[owner];
8         uint256 _balanceTo = _balances[to];
9         require(owner != address(0) && to !=
              address(0));
10        require(_balanceOwner >= value);
11        require(value <= type(uint256).max -
              _balanceTo &&
12            _balanceTo <= type(uint256).max -
                  value);
13
14        _transfer(owner, to, value);
15
16        assert(_balances[owner] == _balanceOwner
              - value);
17        assert(_balances[to] == _balanceTo +
              value);
18
19        return true;
20    }
21    ... // Rest of the smart contract functions
22 }
```

Listing 3: OpenZeppelin's ERC20 Token

Unfortunately these tools lack integrated mechanisms to validate the consistency between the sum of individual balances and the total token supply. In this study, we deliberately omitted such verification procedures (in the form of a contract invariant as presented in Listing 2) in order to maintain the simplicity of the original smart contract. Implementing this feature would require additional functions and commands, such as incorporating an address array in the contract state space and developing a loop-based function to aggregate balances. Such considerable instrumentation of the original contract may change its semantics and, consequently, compromise the result generated by the verifiers. The objective of our investigation is to evaluate the verification tools applied to the original (unchanged) contract in order to analyse the feasibility to employ them in practice.

3.3 The Certora Prover

The Certora Prover (Certora.com, 2024) establishes a significant advancement in the field of formal verification of smart contracts written in Solidity. This industrial-size tool employs sophisticated techniques to rigorously prove the correctness of smart contract implementations. Unlike many of its predecessors, the Certora Prover operates at the level of the Ethereum Virtual Machine (EVM), rather than directly on the Solidity source code. This approach allows for a more comprehensive analysis that accounts for the intricacies of EVM execution.

The tool workflow begins with the translation of Solidity contracts into an intermediate representation that captures the semantics of the EVM bytecode. This representation is then subjected to the analysis of specialized theorem-proving engines based on state-of-the-art SMT solvers. These engines are capable of reasoning about complex logical assertions and arithmetic operations, thus enabling the verification of a wide spectrum of properties.

One of the distinguishing features of the Certora Prover is its support for an extensive range of property types. It can verify generic properties like safety and liveness, and domain-specific properties. Safety properties verify that bad states are unreachable, while liveness properties guarantee that desirable states are eventually reached. And the tool allows for the specification of custom properties through its proprietary specification language, thus enabling developers to express and verify domain-specific requirements.

The Certora Prover user interface is designed with a focus on the developer user friendliness. It provides clear, actionable feedback, including counterexamples when properties fail to hold. This feature significantly aids the debugging process, thus allowing developers to quickly identify and rectify issues in their smart contract implementations.

It is worth noting that the Certora Prover occupies a unique position in the market as one of the few industrial-strength verification tools for smart contracts. While it generally requires a paid license for commercial use, the company occasionally offers free licenses, particularly for academic research or open-source projects.

The tool capabilities extend beyond mere bug detection. It can provide formal guarantees of correctness with respect to specified properties, a crucial feature for high-stakes applications in decentralized finance (DeFi) and other blockchain-based systems where code immutability makes post-deployment fixes challenging and costly.

In summary, the Certora Prover is a powerful addition to the smart contract developer's toolkit, offering rigorous formal verification capabilities that can significantly enhance the reliability and security of blockchain applications. Its EVM-level analysis, coupled with its user-friendly interface and support for diverse property types, positions it as a valuable asset in the ongoing effort to improve the robustness of smart contract ecosystems.

3.3.1 ERC20 formalization for The Certora Prover

The Certora Prover uses an approach similar to solc-verify where a specifically designed language— CVL (Certora Verification Language)—is used to formalize the properties to verify.

The Listing 4 shows an excerpt of the Certora Verification Language (CVL). The code begins by importing auxiliary functions and method specifications for the ERC20 interface (lines 1–4). A formal verification rule for the `transfer()` function of the ERC20 token contract is defined (lines 6–26). Note that, unlike Solidity `assert` and `require`, the CVL syntax for `assert` and `require` does not require parentheses around the expression and the message. The main component is the `transfer()` rule (`integrity_of_transfer()` at line 6), which verifies the correctness of the token transfer operation.

- Environment setup (lines 9–10): The rule defines an environment `e` and sets the token holder as the message sender;
- State caching (lines 12–13): It records the initial balances of the `owner` and the recipient (`to`);
- Transaction execution (line 18): The `transfer()` function is called without the possibility of reverting[3];
- The preconditions at lines 15–16 are introduced in order to avoid Certora (and all other provers) to automatically find edge cases like overflow;
- The postcondition verifies that (lines 23–25):
  - The owner's balance has decreased by the amount transferred; and
  - The recipient's balance has increased by the amount transferred;

In CVL, all integer operations are exact and use the type `mathint` (all integer types are subtypes of `mathint`). In the rule `integrity_of_transfer()`, the function `to_mathint()` converts an integer to a `mathint()` type. This type conversion allows us to specify the conditions for the calculation of the balance without worrying about underflow and overflow scenarios.

This formal verification approach aims to ensure that the transfer function adheres to the expected behavior under various conditions, thus enhancing the reliability and security of the smart contract.

The complete formalization can be found in the Github repository[4].

4 Comparison

This section presents a comprehensive comparison and critical analysis of the four formal verification tools investigated: solc-verify, SMTChecker, VeriSmart and Certora.

We have used the ERC20 token standard as a benchmark for assessing how these four tools handle the formalization and the verification of a real-world smart contract. It is important to emphasize that the ERC20 token contract used as a benchmark in this study is based on a standard that is a widely

[3] In order to allow a function to revert, the suffix `@withrevert` may be added to the function name

[4] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/certora/specs/ERC20.spec`

```
methods {
    function balanceOf(address account) external
        returns (uint256) envfree;
    function transfer(address to, uint256 value)
        external returns (bool);
}

rule integrity_of_transfer(address to, uint256
    value) {
    address owner;

    env e;
    require e.msg.sender == owner && owner != to;

    mathint _balanceOwnerBefore = balanceOf(owner
        );
    mathint _balanceToBefore = balanceOf(to);

    require _balanceOwnerBefore >= to_mathint(
        value);
    require 0 <= _balanceToBefore + to_mathint(
        value) && _balanceToBefore + to_mathint(
        value) <= 2^256-1;

    transfer(e, to, value);

    mathint _balanceOwnerAfter = balanceOf(owner)
        ;
    mathint _balanceToAfter = balanceOf(to);

    assert _balanceOwnerAfter ==
        _balanceOwnerBefore - to_mathint(value),
        "transfer must decrease sender's balance
        by amount";

    assert _balanceToAfter == _balanceToBefore +
        to_mathint(value), "transfer must
        increase recipient's balance by amount";
}
```

Listing 4: OpenZeppelin's ERC20 Token

accepted implementation, typically considered robust and secure for its intended purpose. The core objective of this research is not to assess the inherent correctness or to discover vulnerabilities within this specific ERC20 implementation itself. Rather, the analysis focuses on evaluating the performance, practical applicability, and limitations of the four formal verification tools (solc-verify, SMTChecker, VeriSmart, and Certora Prover) against a well-understood and standardized smart contract. This includes examining criteria such as tool accuracy in verifying specified Hoare logic-based properties (preconditions, postconditions, and invariants), verification time, compatibility with Solidity versions, and to what extent code adaptations are needed for verification. Therefore, our subjects are the tools (not the contracts) and their practical usage. In fact, we are not measuring here the tools effectiveness in finding bugs and such a limitation remains to be addressed in future work. The original ERC20 contract had to be slightly modified before verification: (i) all abstract functions became concrete functions (so that the contract could be compiled); and (ii) we added the pre and postconditions of the `transfer()` function. Our ERC20 benchmark comprises 8 files, 23 functions, ≈450 LoC in flattened form, covering all standard operations.

We evaluate these tools across multiple criteria, including support for current Solidity versions, ability to verify a real-world (almost unchanged) contract, support for local execution, verification time, use of domain-specific languages for modeling, among others.

This evaluation aims to provide researchers and developers with a clear understanding of the capabilities and practical applications of these formal verification tools in the context of Ethereum smart contract development and security assurance.

4.1 solc-verify

The solc-verify tool (Hajdu and Jovanović, 2019) was executed under different conditions compared to the other tools in our study. To facilitate its operation, we utilized Docker for deployment[5]. It is important to note that this approach may have impacted execution time measurements, as Docker imposes a significant computational overhead on a machine with 16GB RAM and a 4.4GHz Intel Core i5 processor. Besides that, it is worth mentioning that we had to downgrade the Solidity version to 0.5.11 to run solc-verify. Despite these limitations, solc-verify was executed according to its documentation and successfully analyzed the entire ERC20 smart contract as shown in Listing 5.

4.2 SMTChecker

The use of SMTChecker (Alt and Reitwiessner, 2018b) was almost straightforward as it did not support `revert` statements associated with error elements (we had to use just `revert()` instead). Since SMTChecker does not model custom error enums, we replaced all revert `ERC20Error(...)` calls with bare `revert()` in the source, but did not affect semantics.

[5]Docker is a platform for running container applications. `https://www.docker.com/`

```
IERC20Errors::[implicit_constructor]: OK
IERC721Errors::[implicit_constructor]: OK
IERC1155Errors::[implicit_constructor]: OK
IERC20::[implicit_constructor]: OK
Context::[implicit_constructor]: OK
IERC20Metadata::[implicit_constructor]: OK
ERC20::[constructor]: OK
ERC20::name: OK
ERC20::symbol: OK
ERC20::decimals: OK
ERC20::totalSupply: OK
ERC20::balanceOf: OK
ERC20::transfer: OK
ERC20::transferFrom: OK
ERC20::allowance: OK
ERC20::approve: OK
No errors found.
```

Listing 5: solc-verify report on ERC20

```
Warning (1218): CHC: Error trying to invoke SMT solver.
  --> src/ERC20.sol:50:9:
   |
50 |         assert(_balanceOwner + _balanceTo == _balances[owner] + _balances[
      to]);
   |
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

Warning (6328): CHC: Assertion violation might happen here.
  --> src/ERC20.sol:50:9:
   |
50 |         assert(_balanceOwner + _balanceTo == _balances[owner] + _balances[
      to]);
   |
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

forge build --force  9.07s user 0.23s system 95% cpu 9.760 total
```

Listing 6: SMTChecker report on ERC20 without adjustments.

Building upon the formalization presented in Section 3.2.1, we made all configurations needed to the `foundry.toml` file to activate the verifier and compile the contract.

At first, SMTChecker did not work. It reported errors and assertion violations, as shown in Listing 6. We discovered that it was related somehow to the function `transferFrom()`.

In a second attempt, we specified the pre and postconditions of both functions `transfer()` and `transferFrom()`, and SMTChecker could verify all properties of `transfer()` and left one postcondition of `transferFrom()` unproven (Listing 7).

In a third attempt, we commented out parts of the function `transferFrom()`. By removing the calls to the internal functions `_spendAllowance(from, spender, value)` or `_transfer(from, to, value)` (or both), all properties of `transfer()` were verified.

Although, at the end, we managed to verify `transfer()` with little changes to the contract, SMTChecker presented an unstable behavior that requires further investigation.

```
Info (9576): CHC: Assertion violation check is
    safe!
  --> src/ERC20.sol:50:9:
   |
50 |         assert(_balanceOwner + _balanceTo ==
     _balances[owner] + _balances[to]);
```

Listing 7: SMTChecker report on ERC20

```
[CHECKER] Integer Over/Underflows
[CHECKER] Division-by-zero
[CHECKER] Suicidal
[CHECKER] Ether-Leaking
- all funcs : 38
- reachable : 10
* [STEP] Generating Paths ... took 0.008807s
- #paths : 428
[INFO] Violate CEI: false
[INFO] msg.sender = this possible: false
* Performing Interval Analysis ... took 0.030432s
Iter : 10 To explore : 8 Explored : 55 Total elapsed : 56.066283
===== Report =====
[1] [IO] line 203, (preBalOwner - value) : proven
[2] [IO] line 204, (preBalTo + value) : proven
[3] [IO] line 256, (preBalFrom - value) : proven
[4] [IO] line 257, (preBalTo + value) : proven
[5] [IO] line 291, (_totalSupply + value) : proven
[6] [IO] line 299, (fromBalance - value) : proven
[7] [IO] line 306, (_totalSupply - value) : proven
[8] [IO] line 311, (_balances[to] + value) : proven
[9] [IO] line 408, ((2  256) - 1) : proven
[10] [IO] line 408, (2  256) : unproven
[11] [IO] line 413, (currentAllowance - value) : proven
============ Statistics ============
# Iter                    : 11
# Alarm / Query           : 1 / 11
- integer over/underflow  : 1 / 11
- division-by-zero        : 0 / 0
- kill-anyone             : 0 / 0
- ether-leaking           : 0 / 0
Time Elapsed (Real) : 71.470870018
Time Elapsed (CPU)  : 71.221194
real    1m11,479s
user    1m10,783s
sys     0m0,462s
```

Listing 8: VeriSmart report on ERC20 - (Verification 1)

4.3 VeriSmart

VeriSmart (So et al., 2020) also required an extra layer of modifications. As VeriSmart could not handle imports, it was necessary to first flatten the contract (to inline the imports). Moreover, the Solidity version had to be downgraded to 0.5.11 due to the VeriSmart compatibility limitation. The downgrade required modifications to the original contract. The contract had to avoid the emission of events and the handling of error-related constraints. We also had to adapt the syntax for mappings and data location specifiers. Specifically, the declaration of mappings had to be adjusted from `mapping(address account => uint256) private _balances`, where the identifier `account` is explicitly used, to `mapping(address => uint256) private _balances`. And we had to replace `calldata` by `memory` (`storage` could be used as well). Finally, we had to type in the constant `115792089237316195423570985008687907853269984665640564039457584007913129639935` instead of using the original expression `type(uint256).max`. After making these adjustments, VeriSmart was successfully executed, yielding the results shown in listings 8 and 9. Notably, VeriSmart exhibited non-deterministic behavior across multiple executions of the same contract analysis, thus generating inconsistent results for identical verification queries. This variability undermines the tool's reliability and raises concerns about the reproducibility of its formal verification outcomes.

A comparison of two smart contract verifications using the VeriSmart tool reveals notable differences in the analysis of potential integer overflows. In the initial verification (Listing 8), the tool reported one unproven integer overflow issue out of eleven queries, with an "Alarm / Query" ratio of 1 / 11. The total real-time elapsed for this analysis was 71.479 seconds. In the subsequent verification (Listing 9), the number of unproven integer overflow issues increased to three, resulting in an "Alarm / Query" ratio of 3 / 11. Interestingly, the second verification was completed in a shorter

```
[CHECKER] Integer Over/Underflows
[CHECKER] Division-by-zero
[CHECKER] Suicidal
[CHECKER] Ether-Leaking
- all funcs : 38
- reachable : 10
* [STEP] Generating Paths ... took 0.009109s
- #paths : 428
[INFO] Violate CEI: false
[INFO] msg.sender = this possible: false
* Performing Interval Analysis ... took 0.031072s
Iter : 10 To explore : 8 Explored : 55 Total elapsed : 57.228998
===== Report =====
[1] [IO] line 203, (preBalOwner - value) : proven
[2] [IO] line 204, (preBalTo + value) : unproven
[3] [IO] line 256, (preBalFrom - value) : proven
[4] [IO] line 257, (preBalTo + value) : unproven
[5] [IO] line 291, (_totalSupply + value) : proven
[6] [IO] line 299, (fromBalance - value) : proven
[7] [IO] line 306, (_totalSupply - value) : proven
[8] [IO] line 311, (_balances[to] + value) : proven
[9] [IO] line 408, ((2  256) - 1) : proven
[10] [IO] line 408, (2  256) : unproven
[11] [IO] line 413, (currentAllowance - value) : proven
============ Statistics ============
# Iter                    : 10
# Alarm / Query           : 3 / 11
- integer over/underflow  : 3 / 11
- division-by-zero        : 0 / 0
- kill-anyone             : 0 / 0
- ether-leaking           : 0 / 0
Time Elapsed (Real) : 60.9739511013
Time Elapsed (CPU)  : 60.771262
real    1m0,982s
user    1m0,372s
sys     0m0,420s
```

Listing 9: VeriSmart report on ERC20 - (Verification 2)

time of 60.982 seconds. Specifically, the potential overflows at line 204, (preBalTo + value), and line 257, (preBalTo + value), which were initially proven to be safe, were marked as unproven in the second run. All reported issues were false alarms.

4.3.1 The Certora Prover

The Certora Prover is fully compatible with the ERC20 smart contract without requiring any modification (except those changes related to abstract functions and the addition of pre and post-conditions of the `transfer()` function).

The process begins with the specification file, as briefly discussed in Section 3.3.1, which is linked to the smart contract.

The Certora Python client is then used to verify the license key[6], compile the contract, and send the relevant data to the Certora servers.

The local portion of this computation is efficient, typically taking just a few seconds to run. The primary delay occurs on the server side, where processing can take a few minutes to several hours, depending on the complexity of the smart contract (Mota et al., 2023). Figure 1 illustrates the web-based Certora Prover report generated for the ERC20 smart contract[7].

4.4 Empirical Analysis

We carried out an empirical investigation to assess the performance attributes of several smart contract verification tools quantitatively. Our approach, research questions, and the outcomes of running Verismart, SMTChecker, solc-verify, and

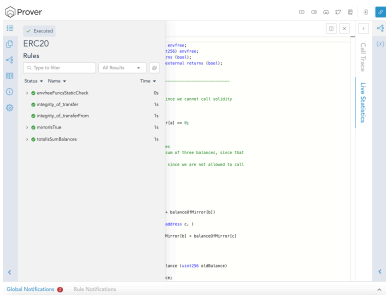

Figure 1. A Certora Prover Report.

Certora on a typical ERC20 token smart contract are described in this part. To guarantee the accuracy and consistency of our measurements, each tool was performed ten times on a laptop running Fedora, version 42, which has a 4.4GHz Intel Core i5 12500h processor and 16GB of RAM.

4.4.1 Research Questions

Our empirical analysis is guided by the following research questions, designed to provide a comprehensive understanding of each tool's performance profile:

RQ1: *Performance Efficiency:* Which tool demonstrates the most efficient performance in terms of total execution time?

RQ2: *Performance Stability:* Which tools exhibit the most consistent and predictable performance across multiple, independent execution runs?

RQ3: *Impact of Execution Environment:* How does the execution environment (local vs. cloud-based) affect the performance efficiency and stability for the Certora tool?

4.4.2 Methodology

Our experiment's main objective is to evaluate the performance of four tools using a common benchmark, which is a standard implementation of the ERC20 token contract. We distinguish five distinct execution configurations for our analysis: Verismart, SMTChecker, solc-verify, Certora (Cloud), and Certora (Local). To gather information on user time, system time, and total elapsed (wall-clock) time, we ran each configuration ten times in a row.

It is important to note the specific execution contexts. Certora was evaluated in two modes: its default client-server architecture, where a local process submits the verification job to a remote cloud service (referred to as **Certora (Cloud)**), and a fully local execution mode (**Certora (Local)**). For the cloud-based mode, we consider the "Job Runtime" as the total wall-clock time from a user's perspective. Additionally, solc-verify was run inside a Docker container due to its setup requirements.

To answer our research questions, we propose a statistical analysis of the collected data.

1. **Descriptive Statistics:** We first calculate descriptive statistics (mean, standard deviation, median, min, max) for the total execution time of each of the five configurations. This provides a high-level overview of performance;

[6] In this work, we utilize the free license key; however, a paid license key, which typically costs around USD 250,000 per year, is generally required.

[7] This web page is accessible at `https://prover.certora.com/output/480394/668e8ce1a5334c23baef672480e91734?anonymousKey=fbf6c50aed1e0039887311b9b27b5eb5c706d863`

2. **Hypothesis Testing for Performance (RQ1 and RQ3):** We perform a one-way Analysis of Variance (ANOVA) test to ascertain whether the observed variations in mean execution times across the five configurations are statistically significant. A p-value below the significance level of $\alpha = 0.05$ would indicate a significant difference;
3. **Hypothesis Testing for Stability (RQ2 and RQ3):** We apply Levene's test for homogeneity of variances to formally evaluate performance consistency. This test identifies whether the variation in execution times differs in a statistically significant way across the configurations.

4.4.3 Results and Discussion

The aggregated results from our 10 experimental runs for each tool configuration are summarized in Table 3. For Verismart, SMTChecker, solc-verify, and Certora (Local), the "Real" time from the raw data is used for analysis. For Certora (Cloud), the "Job Runtime" is used, as it reflects the complete wall-clock time from the user's perspective.

RQ1: Performance Efficiency Based on the mean execution times shown in Table 3, we can establish a clear performance ranking among the five configurations. **solc-verify** is, by a significant margin, the fastest tool with an average execution time of 0.0084s. It is followed by **SMTChecker** (10.15s) and then **Certora (Local)** (17.93s). The cloud-based and slowest tools are **Certora (Cloud)** (49.50s) and **Verismart** (68.19s).

In order to evaluate the performance difference among the five configurations, a one-way Analysis of Variance (ANOVA) was performed. The statistical hypotheses are extended to include the two Certora modes.

Null Hypothesis (H_0) The null hypothesis posits that there is no difference in the mean total execution times across all five groups.

$$H_0 : \mu_{\text{Certora (Cloud)}} = \mu_{\text{Certora (Local)}} = \mu_{\text{Verismart}} = \mu_{\text{solc-verify}} = \mu_{\text{SMTChecker}} \quad (1)$$

Alternative Hypothesis (H_a) The alternative hypothesis posits that at least one of the group means is different from the others.

Given the p-value of 5.5666×10^{-21} from the ANOVA test, which is well below the significance level of $\alpha = 0.05$, we reject the null hypothesis. This confirms a statistically significant difference in the performance of the tool configurations.

RQ2: Performance Stability Performance stability, measured by the standard deviation of execution time, varies significantly. **solc-verify** (SD = 0.0007s) exhibits near-perfect stability. **SMTChecker** (SD = 0.3135s) and **Certora (Local)** (SD = 0.4965s) also demonstrate highly consistent and predictable performance. **Verismart** (SD = 5.0064s) is considerably less stable.

By a large margin, **Certora (Cloud)** shows the lowest stability (SD = 20.5872s). This high variance confirms that its performance is heavily influenced by external factors like network latency and server load, making it the least predictable on a per-run basis.

In order to formally assess these differences, a Levene's test for homogeneity of variances was conducted. The Levene's test yielded an F-statistic of 6.6508 and a p-value of 2.7073×10^{-4}. Since the p-value is significantly less than 0.05, we reject the null hypothesis of equal variances, confirming that the tools do not share the same level of performance stability.

RQ3: Impact of Execution Environment (Certora) The most direct way to answer RQ3 is to compare the "Certora (Cloud)" and "Certora (Local)" results from Table 3. The difference is clear. On average, the local execution (17.93s) is approximately **2.76 times faster** than the cloud-based job runtime (49.50s).

Even more striking is the difference in stability. The standard deviation for local execution (0.4965s) is over **40 times smaller** than that of the cloud execution (20.5872s). This demonstrates that the local environment provides highly predictable and consistent performance, whereas the cloud environment introduces significant variability. The trade-off is clear: while the cloud service may offer more powerful underlying hardware for more complex verification tasks and offloads work from the user's machine, it comes at the cost of significantly lower speed and predictability for standard tasks due to queuing, network, and server-side overhead.

4.5 Discussion

Table 5 summarizes the results of our analysis. The first column shows the criteria used in our comparison. The criteria reflect real-world concerns—version support, minimal code changes, etc.—each vital for seamless integration into developer workflows. In Table 5, $\surd$ indicates full support, $\times$ no support, and $\approx$ partial or conditional support.

The criteria in Table 1 were chosen to reflect key practical concerns for developers and auditors. Current Solidity support is essential for leveraging the latest language features and security protections. The ability to verify the Original structure with minimal changes reduces the risk of introducing new bugs during the verification setup. Local execution and fast execution time are critical for seamless integration into agile development and CI/CD pipelines, while the No DSL for modelling criterion reflects the learning curve and accessibility of a tool.

From Table 5, we can observe that:

- *Current Solidity*: Both solc-verify and VeriSmart support Solidity versions up to $0.5.*$, while SMTChecker and Certora have fuller support for current Solidity versions. This suggests that SMTChecker and Certora may be more up-to-date and suitable for projects using newer Solidity features;
- *Source language*: Certora is the only tool that verifies the contract at the level of the EVM. The other tools

Table 3. Revised Descriptive Statistics for Total Execution Time (in seconds)

Tool Configuration	Mean	Std. Dev.	Median	Min	Max
Certora (Cloud)	49.5000	20.5872	42.0000	18.0000	90.0000
Certora (Local)	**17.9324**	**0.4965**	**17.7470**	**17.5060**	**18.7780**
SMTChecker	10.1531	0.3135	10.0415	9.8430	10.8410
solc-verify	**0.0084**	**0.0007**	**0.0085**	**0.0070**	**0.0090**
Verismart	68.1940	5.0064	70.9335	60.8360	72.1310

Table 4. ANOVA Results for Total Execution Time

Source of Variation	Sum of Squares	df	Mean Square	F-statistic	p-value
Between Groups	12300.99	4	3075.25	91.1120	5.5666e-21
Within Groups	1520.25	45	33.78		
Total	13821.24	49			

verify the contract at the Solidity source code abstraction level;

- *Original structure*: By *original structure* we mean the source code of the contract with only two changes: (i) the removal of abstract functions; and (ii) the addition of the pre and post-conditions of `transfer()`. We regard a minor change either flattening the contract (trivially obtained by inlining the imports) or adding pre and post-conditions to `transferFrom()`). A downgrade to an older version can be regarded as a medium sized change. Only Certora fully satisfies the original structure criterion. All tools required some change. And, in particular, SMTChecker presented an unstable behavior that required us to engage in a few interactions with the tool in order to converge to a verifiable version with minor impact on the semantics (by adding pre and post-conditions to `transferFrom()`).
- *Local execution*: Local execution is supported by all tools except Certora[8]. This could make Certora less convenient for developers who prefer or require a local verification environment;
- *No DSL for modelling*: SMTChecker and VeriSmart stand out because they do not require a domain-specific language (DSL) for modeling the specification (they make use of `require` and `assert` of Solidity). This make them more accessible to developers who are not familiar with specialized modeling languages;
- *Library support*: All tools offer some level of library support, depending more on the Solidity version supported as well as if the smart contract must be in flat mode (VeriSmart) or not (the other tools);
- *Inheritance support*: inheritance support is related to the capabilities of the Solidity version supported;
- *Solidity full support*: Interestingly, none of the tools offer full support to all Solidity features. Certora is the tool that offers the closest full support. Section 4.6 gives more details about this;
- *Execution time*: solc-verify took 0.0084sec; SMTChecker took 9.60sec; VeriSmart took 67.60sec; and Certora took 49.50sec[9]. These execution times give a clear indication of the relative performance of each tool, with solc-verify being the fastest at 0.0084sec, and Verismart being the slowest at 67.60sec. This allows for a direct comparison of their efficiency in terms of execution speed;
- Professional support: Certora is the only tool noted to have professional technical support. This could be a significant advantage for projects requiring robust, production-ready verification;
- *Free license*: Certora is a premium tool that typically requires a paid license to use. However, the company occasionally provides free licenses during specific time-limited bounty competitions, allowing participants to get temporary access to the tool without charge. Certora has offered a completely free license for non-commercial and academic use since March 2025.

In conclusion, each tool has its strengths and weaknesses. SMTChecker appears to be a well-rounded option with good language support and no DSL requirement despite showing some instability. Certora stands out for its professional technical support and comprehensive feature set, despite lacking local execution. solc-verify and VeriSmart seem to lag behind in compatibility with current Solidity versions, but offer solid results in other features.

The choice of tool would depend on specific project requirements, such as the need for local execution, current Solidity version support, or real-world applicability. For critical, production-level projects, Certora might be the preferred choice due to its professional support as described in the work by (Mota et al., 2023). For developers working with newer Solidity versions and preferring local execution, SMTChecker could be ideal, although it does not support large systems. Projects using older Solidity versions might find solc-verify or VeriSmart sufficient for their needs.

[8]Certora stated in February 2025 that the Certora Prover will be made publicly available and locally implemented for basic use.

[9]Although it can take several minutes on the Certora server.

	solc-verify	**SMTChecker**	**VeriSmart**	**Certora**
Current Solidity	$\leq 0.5.*$	$\surd$	$\leq 0.5.*$	$\surd$
Source language	Solidity	Solidity	Solidity	Solidity/EVM
Original structure	$\times$	$\approx$	$\times$	$\surd$
Full ERC20	$\surd$	$\surd$	$\surd$	$\surd$
Local execution	$\surd$	$\surd$	$\surd$	$\surd$
No DSL for modelling	$\times$	$\surd$	$\surd$	$\times$
Library support	$\surd$	$\approx$	$\approx$	$\surd$
Inheritance support	$\approx$	$\surd$	$\approx$	$\surd$
Solidity full support	$\times$	$\times$	$\times$	$\approx$
Execution time	0.0084sec	9.60sec†	67.60sec	49.50sec
Real-world support	$\times$	$\times$	$\times$	$\surd$
Free license	$\surd$	$\surd$	$\surd$	$\approx$

Table 5. Comparison of Solidity Formal Verification Tools

It is worth noting that the lack of full Solidity support across all tools suggests that the field of formal verification for Solidity is still evolving, and developers may need to use a combination of tools for comprehensive coverage.

While this study omits direct defect detection rates, future work will incorporate comparative security benchmarks.

4.6 Real-World Applicability

In 2022, the work (Mota et al., 2023) conducted an extensive investigation into formal verification tools for smart contracts aiming at finding those tools suitable to formally verify current Solidity versions. This investigation covered all tools cited in the scientific literature.

That study revealed significant limitations in the applicability of existing tools to real-world smart contracts. These limitations can be attributed to several factors:

- Version discrepancy: Academic tools often lag behind industry standards, where most of them support Solidity versions from $0.4.*$ to $0.5.*$, while real-world contracts utilize more recent versions;
- Scale and complexity: While academic studies typically focus on simpler contracts like ERC20 tokens (approximately 450 lines of code in their flat version), real-world systems often exceed 5,000 lines of code;
- Advanced language features: Many aspects of Solidity commonly used in practice have not been adequately addressed in the scientific literature. These include inline assembly code, frequent use of Keccak256 constants, and the use of variables declared as interfaces but instantiated as specific contracts;
- Maintenance and updates: The rapidly evolving nature of smart contract development requires continuous updates of the verification tools. This level of maintenance is typically only achievable with corporate backing, as exemplified by Certora.

It is worth noting that there are atypical cases, such as the tools developed by Runtime Verification (e.g. Klab (DappHub, 2024) and Kontrol (RuntimeVerification, 2024)), which are based on the KEVM semantics (Hildenbrandt et al., 2017). Despite being corporate-backed, these tools have shown questionable reliability even for small smart contracts. The only trustworthy solution we are aware of is the ERCx tool (ERCX, 2024), which is basically a fuzz testing based framework. However, these tools focus more on symbolic testing rather than Hoare logic-based formalizations.

These findings highlight the need for more robust and adaptable formal verification tools that can keep pace with the rapid evolution of smart contract development practices.

4.7 Threats to Validity

In this section, we discuss threats to the validity of our study and the steps taken to mitigate them.

Internal Threats:

- Tool Configuration: The performance and the effectiveness of the formal verification tools may be sensitive to specific configurations. In order to mitigate this, we followed the official documentation for each tool and used default settings whenever possible. However, there may still be room for optimization that could affect the results;
- Benchmark Selection: Our use of a single contract as a benchmark may not fully represent the diversity of smart contracts in real-world applications. The choice for the ERC20 token standard was due to its widespread use, manageable size and standardization, thus providing a common ground for comparison. Although some contracts are not even supported by some of these tools, we intend to investigate ERC721, ERC1155, and ERC4626 in the future as well;
- Version Compatibility: The need to downgrade Solidity versions for some tools (e.g., solc-verify and VeriSmart) may have affected the comparison's fairness. We acknowledge this limitation and have clearly stated the versions used for each tool in our analysis;
- Execution Environment: The use of Docker for solc-verify and differences in execution environments may have impacted performance measurements. We have explicitly noted these differences and their potential effects on the results;
- Scope of Evaluation Metrics: A primary limitation of this study is the deliberate omission of defect detection rates as a comparison metric. Our analysis focuses on applicability and developer workflow integration (e.g., Solidity version support, requirement for code changes, local execution), which are crucial for tool adoption. We justify this focus by arguing that a tool's practical appli-

cability is a necessary precondition for its effectiveness to be relevant in a real-world setting. However, by not measuring bug-finding capabilities, our study does not provide a complete picture of the tools' overall utility and security impact.

External Threats:

- Generalizability: Our findings may not generalize to all types of smart contracts or to future versions of Solidity and the tools examined. To mitigate this, we focused on a widely-used standard (ERC20) and provided detailed context for our evaluations;
- Tool Evolution: Rapid development in the field of smart contract verification means that tools may have been updated since our analysis. We have clearly stated the versions used and the date of our evaluation to contextualize our findings;
- Limited Tool Selection: Our study focused on four specific tools, which may not represent the full spectrum of available formal verification solutions for Solidity. We chose these tools based on their prominence and diversity of approaches, but acknowledge that other tools exist;
- Real-World Applicability: The controlled environment of our study may not fully reflect the challenges of applying these tools in real-world development scenarios. We have attempted to address this by discussing real-world applicability separately and noting limitations observed in practice.

To address these threats, we have:

- Provided detailed information about our methodology, tool versions, and execution environments to enable reproducibility;
- Clearly stated the limitations of our study, particularly regarding version compatibility and the scope of our benchmark;
- Discussed real-world applicability separately to highlight the gap between controlled studies and practical application;
- Suggested areas for future work, including expanding the evaluation criteria and developing standardized benchmarks.

While these measures help to mitigate the identified threats, we acknowledge that some limitations remain inherent to the nature of the study. We encourage readers to consider these factors when interpreting and applying our findings.

5 Related work

Prior surveys of smart contract formal verification have focused on taxonomizing techniques and summarizing existing research. Almakhour et al. (2020) comprehensively categorize verification methods (model checking, theorem proving, symbolic execution), identifying key challenges in automation, expressiveness, and scalability, and trade-offs between accuracy and usability. However, their literature review omits solc-verify, SMTChecker, and Certora Prover from detailed analysis and explicitly addresses bug-finding effectiveness.

Garfatta et al. (2021) examine Solidity verification tools by supported qualities and logic types, noting limited support for expressive specifications. They cover solc-verify and VeriSolid but omit SMTChecker and Certora Prover. Both surveys summarize existing findings rather than provide new empirical evaluation.

Our work provides novel empirical comparative analysis by directly evaluating four Hoare logic-based verifiers — solc-verify, SMTChecker, VeriSmart, and Certora Prover — on an ERC20 benchmark. Unlike previous literature reviews, we assess practical adoption criteria: Solidity compatibility, minimal code adaptations, local execution support, verification efficiency, and domain-specific language requirements. We deliberately omit defect detection rates to focus on practical applicability, addressing gaps in real-world evaluation of industrial tools often overlooked in academic analyses.

Bartoletti et al. (2025) compared Solidity versus Move verification, focusing on language design impacts using the Certora Prover and Aptos Move Prover with bank, vault, and price-bet contracts, contributing the first open verification dataset for both platforms. Our work focuses exclusively on Solidity, providing a novel comparison of four Hoare logic-based tools: solc-verify, SMTChecker, VeriSmart, and Certora Prover. While they examined language design, we addressed practical adoption criteria for developers and auditors — Solidity support, code modification needs, execution requirements, and verification time. Using ERC20 as a real-world benchmark and evaluating industry-grade tools like the academically under-analyzed Certora Prover, our work uniquely bridges the academia-industry gap in smart contract verification.

Besbas et al. (2024) provide a literature review exploring three formal methods (model checking, theorem proving, and F* translation) for blockchain verification, with primary focus on the first two approaches and limited attention to F* methods. In contrast, our work offers a novel empirical evaluation of four Hoare logic-based tools for Solidity smart contracts: solc-verify, SMTChecker, VeriSmart, and Certora Prover. Rather than broad conceptual exploration, we systematically assess real-world tools against practical adoption criteria including Solidity compatibility, minimal code adaptation requirements, local execution, verification efficiency, and domain-specific language needs. We directly address the academic community's narrow focus beyond ERC-20 by providing targeted empirical evaluation of the widely-used ERC20 standard as a real-world benchmark. By including industry-grade tools like Certora Prover, which receives limited academic analysis, our study bridges the academia-industry gap in smart contract verification, delivering practical insights into workflows and adoption barriers beyond general formal methods surveys.

The present study meticulously investigated four prominent formal verification tools for Solidity smart contracts: solc-verify, SMTChecker, VeriSmart, and the Certora Prover. These tools were specifically selected due to their adherence to Hoare logic-based approaches, emphasizing formal verification through preconditions, postconditions, and invariants. The research objective was to systematically evaluate

their real-world applicability for developers and security auditors working with production-grade smart contracts. This evaluation was conducted against practical criteria, including compatibility with recent Solidity versions, the ability to verify contracts with minimal adaptations to the original source code, local execution support, verification time efficiency, and the requirement for domain-specific languages (DSL) for property specification. Importantly, this analysis explicitly did not measure bug-finding effectiveness, focusing instead on metrics crucial for tool adoption and integration into development workflows, a recognized gap in the existing literature. The choice of fewer tools for in-depth analysis, compared to the broader landscape of verification methodologies, was therefore a deliberate methodological decision, driven by the study's specific focus on Hoare logic-based formal verifiers and their practical integration challenges. In contrast, diverse verification methodologies beyond this Hoare logic-based focus were acknowledged but not subjected to the same exhaustive empirical evaluation. For instance, KLab (DappHub, 2024) and Kontrol (RuntimeVerification, 2024), which operate at the Ethereum Virtual Machine (EVM) level based on KEVM semantics (Hildenbrandt et al., 2017) and employ symbolic execution, were noted to have limited reliability even for small contracts. Furthermore, these corporate-backed tools prioritize symbolic testing over Hoare logic-based formalizations. Similarly, ERCx (ERCX, 2024), while presented as a trustworthy fuzz testing framework for ERC standard compliance, was characterized by its focus on property testing rather than rigorous formal verification. Manticore (Mossberg et al., 2019), a symbolic execution engine enabling dynamic analysis, was also mentioned as a complementary tool. These alternative tools, despite their relevance to smart contract security, were considered outside the immediate scope of the comparative analysis because their foundational approaches or observed performance issues did not align with the study's specific criteria for evaluating the practical applicability of Hoare logic-based formal verification tools.

Fekih et al. (2025) thorough systematic review highlights important gaps in the formal verification of ERC-based smart contracts and highlights the need for focused research to address changing requirements. According to their review of 19 studies (2019–2023), ERC standards like ERC-721 (non-fungible tokens) and ERC-1155 (hybrid tokens) provide special verification difficulties, such as complicated ownership logic and dynamic state transitions, which are mainly ignored by current tools. Their research draws attention to the academic community's narrow focus outside of ERC-20, but it also subtly emphasizes how crucial thorough ERC-20 verification is—a need that our study successfully fills. Our work directly addresses Fekih et al. (2025)'s call for bridging academia-industry divides by conducting the first comprehensive empirical evaluation of four leading verification tools (including Certora, SMTChecker, and sole-verify) against the ERC-20 benchmark. This evaluation offers crucial insights into practical workflows, industrial adoption barriers, and real-world applicability. Additionally, our examination of ERC-20-specific vulnerabilities, like balance consistency and short address assaults, shows how tools like the Certora prover can accomplish compliance for this widely used standard while also being consistent with their taxonomy of classic and ERC-specific hazards. Although Fekih et al. correctly call for a greater emphasis on more recent ERC standards, our work's detailed analysis of ERC-20 provides a methodological framework for further research, guaranteeing that developers and scholars can build upon a thoroughly tested framework for the most popular smart contract interface.

Our work addresses two critical gaps identified in prior research. First, while existing surveys, as the one produced by Fekih et al. (2023), note the neglect of ERC standards in formal verification literature, we provide the first in-depth evaluation focused specifically on the ERC20 token standard — the most widely adopted smart contract interface. Second, we bridge the academic-industry gap by including the Certora Prover, an industry-grade verification tool that is rarely analyzed by academia despite its growing adoption in enterprise blockchain development. This dual focus on standardized contracts and production-ready tools distinguishes our study from previous comparative analyses.

Liu et al. (2024) took a different angle to the problem of verification. They devised a tool called PropertyGPT that generates the properties to be verified. PropertyGPT leverages on the large language models (LLM) in-context learning in order to encode existing human-written properties (taken from 23 Certora projects) and to generate customized properties to formally verify unknown code. Compilation, static analysis and similarity ranks guide the LLM to produce a compilable and appropriate property. PropertyGPT achieved 80% recall compared to the ground truth. Although this work is not directly related to the verification itself, it automates an important step of the process that currently employs manual labor.

Fekih et al. (2023) conducted a literature survey on smart contracts verification using model checking. They have adopted a different set of criteria than us: the formalism used (Promela, timed automata, transition system, Petri nets, etc), the properties verified (safety, liveness, correctness, security, etc), the system under verification (single or interacting) and the blockchain platform (Ethereum, Bitcoin, Hyperledger Fabric, etc). The model checkers analyzed came from 26 articles in total, where 10 used NuSMV and nuXmv, 3 used SPIN and BIP-SMC, 2 used CPN and the remaining works used other verifiers such as Maude, MCMAS and Helena. One of their main conclusions is that ERC-based contracts have been neglected by the articles analyzed.

Tolmach et al. (2021) provides a comprehensive overview of formal specification and verification approaches for smart contracts, covering various tools and techniques. Our article complements this work by focusing specifically on a comparative evaluation of four prominent Solidity formal verification tools (solc-verify, SMTChecker, VeriSmart, and Certora), providing in-depth insights into their unique features and trade-offs in the verification of a real contract.

Wei et al. (2023) and Happersberger et al. (2023) present a comparative evaluation of automated analysis tools for Solidity smart contracts in a similar approach to our work. However, we delved deeper into the capabilities of the tools across specific criteria, such as support for the current Solidity version, handling of original contract code, and use of a specific

verification language.

While Zhang et al. (2023) do not directly compare formal verification tools, they investigate exploitable bugs in smart contracts, which is a relevant concern that our article aims to uncover through the use of formal verification techniques.

The work of Bartoletti et al. (2024) is focused on developing a benchmark for Solidity verification tools, which could complement the findings of our article by providing a standardized framework for evaluating the tools performance and capabilities.

6 Conclusion

Our work presents a comprehensive comparative analysis of four prominent Solidity formal verification tools: solcverify, SMTChecker, VeriSmart, and Certora. Our evaluation focuses on five key criteria: support for the current Solidity language version, ability to verify the original contract code, handling of full ERC20 contracts, support for local execution, and use of a specific verification language.

The findings of this study provide valuable insights into the unique features and trade-offs offered by each tool. solcverify and Certora stand out for their comprehensive support across most criteria, while SMTChecker and VeriSmart excel in specific areas like local execution and original contract verification, respectively. These insights empower Ethereum developers and security experts to make informed decisions when selecting the most appropriate formal verification solution for their smart contract development and auditing needs.

As future work, we intend to expand our evaluation criteria, possibly considering performance, scalability, and support for more advanced Solidity features. Another avenue for future research is related to incorporating new tools, updates, and other smart contracts as well (e.g. ERC721, ERC1155, and ERC4626). As the Solidity language and formal verification ecosystem continue to evolve, it would be valuable to extend this study to include newer tools or updated versions of the existing ones.

Developing a standardized benchmark, building upon the work of Bartoletti et al. (2024), is another important direction. Creating a standardized benchmark for Solidity verification tools could further enhance the comparative analysis and help developers make even more informed choices. A crucial point to further investigate is certainly considering real-world contracts.

Finally, we also intend to investigate how these formal verification tools can be seamlessly integrated into the Solidity development workflow, including IDE support and CI/CD pipelines, which would further improve their adoption and impact.

Artifacts availability. All data related to the empirical evaluation of the four tools used in this work are available in this URL: https://zenodo.org/records/17107636.

Acknowledgements Alexandre Mota would like to thank CNPq for grant number 300263/2025-2 and Lindy Labs for introducing him to real-world smart contracts.

References

Almakhour, M., Sliman, L., Samhat, A. E., and Mellouk, A. (2020). Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:101227.

Alt, L. and Reitwiessner, C. (2018a). SMT-based verification of solidity smart contracts. In *International symposium on leveraging applications of formal methods*, pages 376–388. Springer.

Alt, L. and Reitwiessner, C. (2018b). SMT-based verification of solidity smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, pages 376–388. Springer.

Ante, L. (2020). Smart contracts on the blockchain – a bibliometric analysis and review. *SRPN: Information Technology (Topic)*.

Bartoletti, M., Crafa, S., and Lipparini, E. (2025). Formal verification in solidity and move: insights from a comparative analysis. *arXiv preprint arXiv:2502.13929*.

Bartoletti, M., Fioravanti, F., Matricardi, G., Pettinau, R., and Sainas, F. (2024). Towards benchmarking of solidity verification tools. In *FMBC@CAV*.

Besbas, A., Ailane, A., Kahloul, L., Slatnia, S., and Bourekkache, S. (2024). On the formal verification of smart contracts and blockchain: Challenges and future directions. In *2024 4th International Conference on Embedded & Distributed Systems (EDiS)*, pages 213–217. IEEE.

Cadar, C., Godefroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. *2011 33rd International Conference on Software Engineering (ICSE)*, pages 1066–1071.

Certora.com (2024). The Certora Prover. `https://www.certora.com/`.

DappHub (2024). KLab: A tool for generating and debugging proofs in the K Framework. `https://github.com/dapphub/klab`.

de Moura, L. M. and Bjørner, N. S. (2009). Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods*.

ERCX (2024). ERCx: Property testing for ERC tokens. `https://ercx.runtimeverification.com/`.

Ethereum.org (2024). ERC-20 Token Standard. `https://ethereum.org/en/developers/docs/standards/tokens/erc-20/`.

Fedyukovich, G., Prabhu, S., Madhukar, K., and Gupta, A. (2018). Solving constrained horn clauses using syntax and data. *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9.

Fekih, R. B., Lahami, M., Bradai, S., and Jmaiel, M. (2025). Formal verification of erc-based smart contracts: A systematic literature review. *IEEE Access*.

Fekih, R. B., Lahami, M., Jmaiel, M., and Bradai, S. (2023). Formal verification of smart contracts based on model checking: An overview. In *2023 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 1–6. IEEE.

Garfatta, I., Klai, K., Gaaloul, W., and Graiet, M. (2021). A survey on formal verification for solidity smart contracts.

In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, ACSW '21, New York, NY, USA. Association for Computing Machinery.

Hajdu, Á. and Jovanović, D. (2019). solc-verify: A modular verifier for solidity smart contracts. In *Working conference on verified software: theories, tools, and experiments*, pages 161–179. Springer.

Happersberger, V., Jäkel, F.-W., Knothe, T., Pignolet, Y.-A., and Schmid, S. (2023). Comparison of ethereum smart contract analysis and verification methods. In *European Symposium on Research in Computer Security*, pages 344–358. Springer.

Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., and Roşu, G. (2017). Kevm: A complete semantics of the ethereum virtual machine.

Kulik, T., Dongol, B., Larsen, P. G., Macedo, H. D., Schneider, S., Tran-Jørgensen, P. W. V., and Woodcock, J. (2021). A survey of practical formal methods for security. *Formal Aspects of Computing*, 34:1 – 39.

Le Goues, C., Leino, K. R. M., and Moskal, M. (2011). The Boogie verification debugger (tool paper). In *Software Engineering and Formal Methods: 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings 9*, pages 407–414. Springer.

Liu, Y., Xue, Y., Wu, D., Sun, Y., Li, Y., Shi, M., and Liu, Y. (2024). PropertyGPT: LLM-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580*.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., and Dinaburg, A. (2019). Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE.

Mota, A., Yang, F., and Teixeira, C. (2023). Formally verifying a real world smart contract. *arXiv preprint arXiv:2307.02325*.

Moura, L. d. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

RuntimeVerification (2024). Kontrol formal verification tool. `https://docs.runtimeverification.com/kontrol`.

So, S., Lee, M., Park, J., Lee, H., and Oh, H. (2020). VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE.

Soliditylang.org (2024a). SMTChecker. `https://docs.soliditylang.org/en/latest/smtchecker.html`.

Soliditylang.org (2024b). Solidity language. `https://docs.soliditylang.org/en/v0.8.26/`.

Szabo, N. (1997). The idea of smart contracts. *Nick Szabo's papers and concise tutorials*, 6(1):199.

Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. (2021). A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):1–38.

Wei, Z., Sun, J., Zhang, Z., Zhang, X., Li, M., and Zhu, L. (2023). A comparative evaluation of automated analysis tools for solidity smart contracts. *ArXiv*, abs/2310.20212.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger.

Zhang, Z., Zhang, B., Xu, W., and Lin, Z. (2023). Demystifying exploitable bugs in smart contracts. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 615–627.