Formal Verification in Solidity and Move: Insights from a Comparative Analysis

Massimo Bartoletti
University of Cagliari, Italy

Silvia Crafa
University of Padova, Italy

Enrico Lipparini
University of Cagliari, Italy

Abstract

Formal verification plays a crucial role in making smart contracts safer, being able to find bugs or to guarantee their absence, as well as checking whether the business logic is correctly implemented. For Solidity, even though there already exist several mature verification tools, the semantical quirks of the language can make verification quite hard in practice. Move, on the other hand, has been designed with security and verification in mind, and it has been accompanied since its early stages by a formal verification tool, the Move Prover. In this paper, we investigate through a comparative analysis: 1) how the different designs of the two contract languages impact verification, and 2) what is the state-of-the-art of verification tools for the two languages, and how do they compare on three paradigmatic use cases. Our investigation is supported by an open dataset of verification tasks performed in Certora and in the Aptos Move Prover.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Smart contracts, Solidity, Move, Verification, Blockchain

Digital Object Identifier 10.4230/OASIcs.FMBC.2025.3

Supplementary Material *Dataset*: `https://github.com/blockchain-unica/solidity-vs-move-verification`, archived at `swh:1:dir:8227d479ef035f889cd97557b058dcae2d2f9bd8`

Funding *Massimo Bartoletti*: Partially supported by project SERICS (PE00000014) and PRIN 2022 DeLiCE (F53D23009130001) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.
Silvia Crafa: Supported by the National Recovery and Resilience Plan (NRRP) Project "Securing sOftware Platforms - SOP", CUP H73C22000890001.
Enrico Lipparini: Supported by project PRIN 2022 DeLiCE (F53D23009130001) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU.

1 Introduction

Due to the immutability of the code after deployment and the huge amount of economic assets managed, ensuring the correctness of smart contracts is a crucial task. Attacks exploiting code vulnerabilities and wrong implementations of the business logic are estimated to have caused over $6 billion of losses [13], creating a huge demand for safer and verifiable code.

Solidity, the most adopted smart contract language, presents semantical quirks that make contract implementation quite error-prone, and that highly complicate the verification process. In order to address this issue, several bug-detection tools have been developed [34, 41], as well as some verification tools, that vary in scope, specification language, and level of abstraction. Most notably, SolCMC [1], shipped with the Solidity compiler, and the Certora Prover [15], developed for auditing.

Move is a more recent smart contract language, originally developed for the Diem/Libra blockchain and later adopted by Aptos, SUI and IOTA. Designed with verifiability in mind, Move has been accompanied by a formal verification tool [42] since its early development.

In this work, we investigate how differences in the design of Solidity and Move (in the Aptos dialect) affect verifiability. We base our study on a comparative analysis of a small set of paradigmatic use cases, each evaluated against a range of representative properties. These properties span from low-level aspects, such as function specifications and state invariants, to more high-level ones that characterize the business logic of the contract. For each property, we study the ground truth in Solidity and in Move, and we write, whenever possible, the corresponding formal specifications in the Certora Verification Language and in the Move Specification Language. We focus, in particular, on properties that exhibit discrepancies in ground truths, expressibility, or verifiability. The results of our analysis offer relevant insights about the following research questions:

RQ1) What is the impact of different features of Solidity and Move on verification?
RQ2) What is the state-of-the-art of verification tools for Solidity and Move, and which kind of properties are they currently able to verify?

As an additional contribution, we have developed a **public dataset**[1] (the first of this kind) that serves as a basis of an experimental – and extensible – comparison between the Certora and the Aptos Move specification languages and verifiers.

Structure. The paper starts in Section 2 with an overview of Solidity and Move, and their respective verification tools. Section 3 presents our methodology and discusses the choices of tools, use cases and properties. In Section 4 we present the results of our comparative analysis, addressing RQ1 and RQ2 in Section 4.1 and in Section 4.2, respectively. Finally, in Section 5 we summarize our findings, discuss limitations, and outline future work.

2 Background

In this section we overview the main features of the two languages and of the two verification tools considered in our experimental comparison. In particular, we focus on key design choices of the underlying intended blockchain's model, since it has an impact on both the specification and the verification of contracts' properties.

2.1 Contract languages: Solidity *vs.* Move

From the perspective of smart contract programming, a blockchain is best understood as an asset-exchange state machine, in which the state keeps track of the assets owned by each account, and every transaction contributes to a state transition, possibly creating new assets or exchanging assets among accounts. In Solidity, there are two kinds of accounts: externally owned accounts (EOAs) and contract accounts. The state of the asset-exchange machine can be seen as a map that associates each EOA with a balance of native assets owned by the account (e.g., ETH in Ethereum), and each contract account with a balance and a *storage*, which contains variables and data structures that define the contract state. Differently, in Move the state of the machine can be seen as a map from accounts to the assets owned by them. Assets (called *resources* in Move) are encoded by struct datatypes that enjoy linear

[1] `https://github.com/blockchain-unica/solidity-vs-move-verification`

Listing 1 Simplified Solidity code for the Bank case study

```
contract Bank {
  mapping (address => uint) credits;
  function deposit() payable {  credits[msg.sender] += msg.value;  }
  function withdraw(uint amount) {
    credits[msg.sender] -= amount; payable(msg.sender).transfer(amount);
  }
}
```

Listing 2 Simplified Move code for the Bank case study

```
module bank {
  struct Bank { credits : SimpleMap<address, Coin> } //  resource definition
  fun init(account : &signer) {
    let bank = Bank { credits : simple_map::new() }; // create a resource
    move_to(account, bank); // now signer owns a bank
  }
  fun deposit(sender : &signer, owner : address, amount : u64)  {
    let bank = borrow_global_mut<Bank>(owner); // borrow the resource
    let to_deposit = coin::withdraw(sender, amount); // get sender's coins
    let credit = map::borrow_mut(&mut bank.credits,address_of(sender));
    coin::merge(credit, to_deposit); // increase credit by merging coins
  }
  fun withdraw(sender : &signer, owner : address, amount : u64) {  ... }
}
```

semantics, i.e., a static type system ensures that resources are never duplicated or lost. The main difference between Solidity and Move is the representation and accounting of assets. In particular, accounts in Solidity can only explicitly own native assets, while in Move they can own arbitrary resources. This has a relevant impact, since most real-world contracts involve the creation and exchange of *user-defined* assets, e.g. to represent utilities or market shares in DeFi protocols. Representing and handling user-defined assets in Solidity requires a suitable encoding in the smart contract, while in Move all assets are dealt uniformly.

To exemplify, consider the simplified Solidity code in Listing 1, which encodes a simple bank contract. Once deployed on the blockchain, the global store keeps track of the bank's *balance*, i.e. the amount of ETH associated to the contract account. The contract stores in a variable the `credits` (a *number* that *represents* ETHs) associated to each bank's client. When an account (`msg.sender`) invokes the function `deposit` sending a given amount (`msg.value`) of ETH, the effect is twofold: the ETHs are transferred to the contract's balance, and the sender's additional credit is registered. The `withdraw` function first decreases the number of credits and then transfers the amount of ETH to the sender. The corresponding Move code is shown in Listing 2. The code defines a `bank` module, which relies on two types of *resources*: the `Coins` provided by the underlying platform, and the user-defined `Bank` data structure. The contract is initialized by the function `init`: the signer of the transaction initializes a new, empty, `Bank` and registers its ownership in the global store. When an account invokes the function `deposit`, he takes the role of the `sender` (i.e., the transaction's signer), and the function code executes three steps: (i) the `Bank` resource is borrowed from the address of its `owner`, (ii) an `amount` of `Coins` are borrowed from those owned by the `sender`, and (iii) they are merged with those already registered in the `Bank`'s `credits` map.

Despite its simplicity, this example already shows some differences between the two languages in terms of asset management. In Move, resources are first-class citizens, with properties such as linearity statically guaranteed by the type system. By contrast, in Solidity the user-defined assets must be carefully handled at the contract level (e.g., the logic of the `credits` map must correctly match the flow of ETH), which may be a source of critical bugs.

We will discuss other features and differences between Solidity and Move in Section 4, where they will be instrumental in addressing RQ1 about the impact of language design on smart contract verification.

2.2 Formal verification tools for Solidity and Move

For Solidity, there exist several bug detection tools [18] as well as several verification tools [34]. The two main verification tools are SolCMC [1], shipped with the Solidity compiler, and the Certora Prover [15]. Other verification tools, including SmartACE [37], SmartPulse [31], Solvent [7], VeriSolid [22], VerX [27], and Zeus [16], target various verification aspects, each tool having its own specification language, level of abstraction, and limitations. In this work, we focus on Certora (see Section 3), whose verification language (CVL) [9] features two ways of expressing contract properties: *invariants*, which represent conditions that must remain true across contract transitions, and *rules*, which are a flexible way to specify more general conditions on possible contract transitions. CVL rules can arbitrarily combine requirements on the contract state, calls to contract functions (possibly, leaving some call parameters partially specified), and assertions on the states reached upon these calls.

The Move language, since its early stages, has been tightly coupled and integrated with the Move Prover (MVP): they have been developed and maintained together, and the MVP is intended to be used routinely during smart contract development, likely to an advanced type checker. The Move Prover specification language (MSL) [4, 38] features different ways of expressing properties: function specifications (in terms of pre- and post- conditions), and invariants on functions, on struct datatypes, on global states, and on state transitions. Besides bug detection tools [30], we are only aware of another tool that addresses formal verification of Move contracts, VeriMove [21], built upon the Solidity counter-part VeriSolid.

As an example, consider the property "*after a successful* `deposit`*, the credits of the sender are increased exactly by the amount of tokens deposited*". In CVL (Listing 3), the rule first specifies the call environment `e` (which includes the transaction parameters `msg.sender` and `msg.value`), and stores the sender's credit before the call to `deposit()` in the variable `old_value`. Then, the rule calls `deposit()` with environment `e`, and checks whether the sender's credits after the call have been increased by the amount sent. In MSL (Listing 4), the property is expressed as a function spec (i.e., a specification targeting a single function), in terms of pre and post conditions. The variables tagged with `post` refer to the values of the expressions *after* the call to `deposit`.

Listing 3 Specification of `bank/deposit-assets-credit` in CVL

```
rule deposit_assets_credit {
  env e; // environment variables of the call
  address addr_sender = e.msg.sender; // transaction sender
  mathint amount = e.msg.value; // amount of ETH tokens sent by sender to contract
  mathint old_value = currentContract.credits[addr_sender];
  deposit(e); // perform a successful call to deposit
  mathint new_value = currentContract.credits[addr_sender];
  assert new_value == old_value + amount; // verification condition
}
```

Listing 4 Specification of `bank/deposit-assets-credit` in MSL (simplified)

```
spec bank_addr::bank {
  spec deposit {
    let addr_sender = signer::address_of(sender);
    let old_credits = global<Bank>(owner).credits;
    let old_value = simple_map::spec_get(old_credits, addr_sender).value;
    let post new_credits = global<Bank>(owner).credits;
    let post new_value = simple_map::spec_get(new_credits,addr_sender).value;
    ensures new_value == old_value + amount; // verification condition
  }
}
```

3 Methodology

We now detail the methodology we adopted for our comparative analysis, explaining the choices of the verification tools, use cases, and properties, and how we have built our dataset.

3.1 Verification tools

Given the variety of verification tools available, particularly for Solidity, doing an extensive comparison of all these tools lies beyond the scope of this work. We focus on the Certora Prover for Solidity and the Aptos MVP for Move. The choice of the Aptos MVP is straightforward, as it is, to the best of our knowledge, the only supported version of the Prover at the time of writing, which has furthermore been used to formally verify large Move libraries, including the entire Aptos smart contract layer [26]. We exclude VeriMove [21] as it only supports a strict subset of the language. For Solidity, the variety of available tools is broader. While no single tool strictly outperforms all others in every aspect, we choose the Certora Prover since it is the tool most used in real-world settings for the verification of complex properties. We will nonetheless explicitly mention other tools capable of addressing properties beyond the scope of the two selected tools, whenever applicable. In the following, we will refer to the two tools just as Certora and Move Prover (or MVP). We remark that Certora and MVP have been designed with different goals. In Move, specification and development go side-by-side. Certora, on the other hand, is more oriented to the ex-post analysis of contracts and is primarily used for auditing [11]: consequently, CVL is designed to support the verification of complex properties without requiring modifications to the contract code (e.g., updating ghost variables at given program points). Despite these differences, applying the state-of-the-art tools to a common benchmark is crucial to answer our research questions, namely which properties can be verified in the two languages at the time of writing (RQ2), and how the choice of the contract language affects the quality of the verification process (RQ1).

3.2 Use cases

In the selection of the verification use cases, we identify three paradigmatic smart contracts with increasing level of complexity and exhibiting a rich spectrum of features: a `bank` contract (already described in Section 2.1), a `vault` contract, and a `price-bet` contract.

The `vault` contract implements a security mechanism to prevent an adversary who has stolen the owner's private key from stealing their tokens. Upon creation, the owner specifies its private key, a recovery key, and a wait time. The contract has the following entry points:

- `receive(amount)`, which allows anyone to deposit tokens into the contract;

- `withdraw(receiver, amount)`, which allows the owner to issue a withdraw request, specifying the receiver and the desired amount;
- `finalize()`, which allows the owner to finalize the pending withdraw after the wait time has passed since the request;
- `cancel()`, which allows the owner of the recovery key to cancel the pending withdraw request during the wait time.

The `price-bet` contract implements a bet on a future exchange rate between two tokens. To create the contract, the owner specifies: itself as the contract owner; the initial pot, which is transferred from the owner to the contract; an oracle, i.e. a contract that is queried for the exchange rate between two given tokens; a deadline; a target exchange rate, which must be reached in order for the player to win the bet. The contract has the following entry points:

- `join()`, which allows a player to join the bet. This requires the player to deposit an amount of native cryptocurrency equal to the initial pot;
- `win()`, which allows the player to withdraw the whole contract balance if the oracle exchange rate is greater than the bet rate. This action is disabled after the deadline;
- `timeout()`, which can be called by anyone after the deadline, and transfers the whole contract balance to the owner.

We implement each use case in Solidity and in Aptos Move, ensuring that these implementations remain as close as possible. The verification of these use cases requires to deal with properties featuring several aspects, such as: key-value maps, access control, time constraints, contract-to-contract calls, and transaction-ordering dependencies.

3.3 Properties

For every use cases, we consider an extensive set of properties, ranging from low-level properties that only target single contract functions, to more high-level ones that characterize the global behaviour of the contract. Our choice of properties is based on breadth and diversity (in terms of language features involved, abstraction level, temporal logic structure). Overall, we end up with 66 properties. Aiming at generality, and potentially including also properties that cannot be expressed in the considered tools, we write properties in natural language. We then encode each property, whenever possible, as CVL and MSL specifications. Often, this translation involves adding suitable low-level technical assumptions, to make the specification aligned with the spirit of the corresponding natural language property. As an example, in Move, users may have a frozen coin store that prevents them from receiving tokens; in such a case, even if the natural language property does not mention such aspect, we consider adding such low-level technical assumptions as part of the translation process. Furthermore, coherently with most verification tools, we neglect transaction fees. We then manually annotate the expected truth value in Solidity and Move. Finally, we run the provers and take note of their output. We end up, for each use case, with a sheet consisting of four main columns for each property row: two columns for the ground truths, and two for the provers results. We enrich the table with additional columns containing notes on: the class of the property, the expected truth values, the formal specifications, and the provers outputs.

4 Comparison

Based on our dataset, we now present our comparative analysis. Building upon the analysis of each property, we elaborate our findings to construct an organized knowledge that extends beyond our choice of use cases. In particular, we focus on properties where discrepancies arise

between verification in Solidity and Move. These properties serve as illustrative examples for a broader discussion of the fundamental differences in the verification of the two languages. Our observations can be grouped as follows: properties whose ground truths disagree; properties that trivially hold in one language but not in the other; properties not expressible in one or both specification languages; properties expressible but not verifiable by one or both tools. These four cases are not necessarily independent of one another, but they help to better identify the primary causes of discrepancy. In the first two groups, the discrepancy specifically depends on the contract languages, while, in the latter two, it depends more on the specification language and prover functionalities. We accordingly organise this part into two subsection: Section 4.1 focuses on the impact of the contract languages, while Section 4.2 focuses on the impact of the specification languages and on the provers functionalities.

4.1 Impact of the contract language

Resource preservation. As observed in Section 2, Move enforces asset integrity by ensuring that assets cannot be duplicated but only *moved* between owners; by contrast, Solidity – except that for native tokens (ETH) – requires the management of assets to be implemented at a contract level. For example, in the `bank` use case, the `credits` are rendered in Move as a map from `address` to `Coin` (that *are* the actual assets), while in Solidity they are a map from `address` to `int`. This means that the Solidity code merely *tracks* the assets deposited by each user. However, implementation bugs can lead to a mismatch between the assets controlled by the contract and the overall amount of user credits, assigning more or fewer credits than they are entitled to. This significantly impacts the specification and verification of properties. First of all, in MSL, since credits *are* assets, such properties are implied by properties that concern assets. For example, in MSL the specification of the property

`bank/deposit-assets-credit`: "*after a successful deposit of n tokens, the credits of the sender are increased by n*"

is exactly a sub-specification of the property

`bank/deposit-assets-transfer`: "*after a successful deposit of n tokens, n tokens pass from the control of the sender to the control of the contract.*"

In CVL, by contrast, these two properties are disjoint, and it is possible – in the presence of bugs related to the handling of credits – for the former to hold while the latter is violated. This shows that, to cover the same set of properties, Solidity requires a greater number of specifications than Move. Moreover, in Move, certain properties concerning credits trivially hold, while, in Solidity, they may be hardly verifiable, or even unexpressible. For example:

`bank/credits-leq-balance`: "*the assets controlled by the contract are (at least) equal to the sum of all the credits*"

trivially holds in Move, where `credits` coincide with the deposited assets, but not in Solidity, where `credits` just represent the deposited assets. In general, verifying such kind of properties is quite challenging, as they require to reason about quantities depending on an unbounded number of users.

Access control and ownership. Most smart contracts implement access control mechanisms to ensure that certain actions can only be performed by certain users under certain conditions. A typical check is that some resources can only be updated by functions called by the contract owner. Move inherently supports this kind of check: it suffices that all the functions that update the resource borrow it through a signer address. This is because, in Move, a resource

can only be referenced through the address of its resource owner. This is a security pattern in Move to reduce the risks of access control errors [3]. In Solidity, instead, resource ownership is not a native notion, so it must be encoded by the contract logic. In particular, in order to implement the check above, the contract must first record the owner address in a variable, and each sensitive function must require that the transaction sender and the owner coincide. Forgetting even a single check can lead to vulnerabilities, as in the Parity Wallet hack, where the absence of such check in a function enabled the attacker to become the owner and steal all the contract funds [24]. In our dataset this difference in behaviour can be observed, e.g., for the property

`vault/finalize-revert`: "*a call to* `finalize()` *aborts if the sender is not the owner.*"

In CVL, we need to explicitly check that the address of the sender is equal to the `owner` field, while, in MSL, since the function directly accesses the `Vault` struct owned by the sender, and the *owner* is not determined by the value of a variable but by the address that owns the resource, then the property trivially holds, being enforced by the language. Another typical check is that some addresses used by the contract (e.g., its owner) do not change throughout the contract lifespan (e.g., `vault/owner-immutable`). In Solidity, it is possible to enforce that by declaring the addresses as `immutable`. In such a case, the property is directly enforced by the Solidity compiler, without having to resort to verification. Enforcing the same check in Move is less straightforward. A method is to record the concerned addresses as fields of some struct, and then verify with the MVP that these fields are invariant.

Assets transfer. Solidity and Move render assets and their transfers differently, leading to different techniques for expressing and verifying properties related to them. In Solidity, while there is a clear dichotomy in how the native asset (i.e., ETH) and user-defined assets (e.g., ERC20 tokens) are handled, in both cases transfers are rendered as contract calls. The outcome of a contract call depends on whether the callee is an externally owned account (EOA) or a contract account. When the callee is an EOA, the transfer is guaranteed to succeed, whereas for contract accounts the effect of the call depends on the implementation of the function handling the call. For instance, assets may be returned to the caller if the call reverts, or they may be forwarded (either in full or in part) to other accounts if the function is designed to do so and has enough gas. Therefore, properties about asset transfers should either discriminate between EOAs and contract accounts, or add assumptions about the implementation of the receiver function. However, the first choice is not always viable, as detecting whether an address is an EOA or a contract account (either at contract or specification level) is possible only in limited cases [25, 8]. The second choice is problematic as well, since if the assumptions are false then the property may be violated at runtime. Unlike Solidity, Move offers linguistic primitives for transferring ownership of resources, enabling a more disciplined modelling of asset transfers. This reduces the effort required to incorporate the necessary assumptions when encoding properties. In our dataset, we have observed this, e.g., in the property

`vault/finalize-not-revert`: "*a* `finalize()` *transaction sent by the contract owner, in state REQ, and after the wait time has passed, does not abort*"

which holds in Move but not in Solidity, since the transfer may fail when the receiver is a contract. Furthermore, also

`vault/finalize-assets-transfer`: "*after a successful* `finalize()`*, a given amount of assets pass from the contract to the* `receiver`"

holds in Move but not in Solidity since, if the receiver is a contract, the assets can immediately be transferred to another address through the fallback function.

Function dispatching. Solidity features a form of dynamic dispatching, in that the compiler does not always know, for a contract-to-contract call, the code that will be executed in the callee. This poses significant challenges to verification. Indeed, to avoid unsoundness, verification tools must assume that contract-to-contract calls can execute arbitrary code, which easily leads to false negatives. In order to address the issue, Certora allows users to specify a set of possible implementations of the callee, and verify the caller against each of them [12]. This technique can require considerable effort, and does not resolve the underlying unsoundness issue. Move, on the other hand, features static dispatching, i.e. the compiler (and, consequently, the verifier) know exactly the code that will be executed in the callee. In particular, Move does not support inheritance nor any form of method redefinition. We have observed the impact of these different dispatching designs, e.g., in

`price-bet/win-revert`: "*a* `win` *transaction aborts if the oracle exchange rate is smaller than the bet exchange rate.*"

In Certora, verifying the property requires the user to explicitly instruct the verifier to resolve the call with a given oracle implementation: leaving that unspecified would make verification fail. In practice, many Solidity contracts are written in a way that makes it impossible to predict the actual implementations of the callees (e.g., Solidity contracts using ERC20-compatible tokens usually define only their interface).

Other features. *Immutability.* In Solidity, the `immutable` keyword allows to enforce that certain variables cannot change value throughout the whole lifespan of the contract, making certain properties (e.g., the above-mentioned `vault/owner-immutable`) enforced by the Solidity compiler. In Aptos Move, since an equivalent modifier does not seem to be available, such properties have to be explicitly verified with a prover.[2]

Self-destruct. In Solidity, contracts can receive native tokens at any time through the *self-destruct* method. This requires additional precautions during implementation to prevent funds from getting locked in the contract. For example, our Solidity implementation of the `bank` use case allows users to withdraw only the funds corresponding to their credits (i.e., funds that have been previously deposited). In contrast, funds received via self-destruct cannot be withdrawn from the contract and remain locked. This is not the case in Move, as no equivalent of the self-destruct method exists. For example, the property

`bank/no-frozen-assets`: "*if the contract controls some assets, then it is always possible to transfer them to some user*"

holds in Move, but not in Solidity, since the contract only allows creditor to withdraw the assets they have deposited, but does not provide any function to transfer funds received via self-destruct, resulting in funds getting stuck in the contract.

Necessary technical assumptions. As discussed in Section 3, the translation of properties written in natural language to formal specification often requires the addition of low-level technical assumptions. Here, we report the cases that we have observed in our experiments.

Accepting incoming transfers. As observed in the "*Assets transfer*" paragraph, properties concerning the transfer of assets may need further assumptions on the receiver. In Move, the only technical assumption we had to add in our dataset is that the `CoinStore` of the

[2] Note that, in SUI Move, it is possible to define *frozen objects* (i.e. objects that cannot be modified nor moved). It does not seem possible to define *frozen fields* of an object, though.

receiving address is not `frozen`. In Solidity, one sufficient condition that can be used when the receiver equals to the transaction sender is that the sender is an EOA. Although this could be encoded in CVL by requiring that `e.msg.sender==e.tx.origin`, the Certora prover does not use this additional assumption, leading to a false negative. This is the case, e.g., of:

`bank/withdraw-assets-transfer`: "*after a successful* `withdraw(amount)`*, exactly* `amount` *units of T pass from the control of the contract to that of the sender*"

Other conditions, such as ensuring that the receiver does not fail or does not perform further calls, do not appear to be expressible in CVL.

Coin-to-FungibleAsset. Aptos has recently introduced a "*Fungible Asset*" (FA) standard [2] that extends the *Coin* standard, enabling automatic migration from Coin to FA by default. This automatic migration can make certain properties concerning the transfer of Coins violated, since Coins are not preserved (but migrated to FA). This is the case, e.g., of:

`bank/deposit-revert`: "*a transaction* `deposit(amount)` *aborts if* `amount` *is greater than the T balance of the transaction sender.*"

In order to verify such properties, it is necessary to disable the automatic migration.

Sender is not the contract. In Solidity, it is possible that a contract calls itself. In certain cases, it may be necessary to assume that this is not the case, as, otherwise, certain properties might either not hold, or be unverifiable in practice. For example, `bank/deposit-assets-transfer` specifies that, after a successfull deposit of n tokens, the balance of the sender is decreased by n. While this property is true without further assumptions (since the specific `Bank` contract cannot call itself), in Certora the verification will fail without adding the assumption that the sender is not the contract. This is because verification tools usually over-approximate the set of possible executions, thus considering also the impossible case in which the contract calls itself.

4.2 Impact of the specification language and prover functionalities

We now consider different classes of properties and discuss how (and whether) they can be expressed in the two specification languages. The organization in classes has not to be intended as a formal taxonomy, rather as a schematic way to present our findings.

Function specs. We denote by "function spec" properties that specifically target a given function. We divide these properties into "success conditions", which characterize the conditions under which a function aborts or not, and "post-conditions", which express properties regarding the state after the call, assuming that the call has not aborted. The Move Prover has an ad-hoc specification format for function specs. In Certora, function specs can be expressed as rules that explicitly mention the function being called, and using `requires` statements for pre-conditions, the expression `lastReverted` for checking abort conditions, and the statement `assert` for post-conditions. Listing 3 and Listing 4 presented in Section 2.2 are examples of function specs in CVL and MSL, respectively. Both tools perform well over properties of this kind in our dataset.

State invariants. We denote by "state invariants" properties of the form "*for every reachable state* s*, it holds that* P(s)", where P(s) is a property that only mentions variables in the state s. In Move, state invariants can be proved in two ways: either using a *struct invariant* spec, in case an invariant only deals with a single structure (e.g., in any state, the vault state is IDLE or REQ, i.e. `vault/state-idle-req-inter`), or, otherwise, using a *global invariant*

spec (e.g., the owner and the recovery keys are distinct, i.e. vault/keys-distinct). In Certora, there is a common way to write invariants. Both tools perform well over properties of this kind on our dataset.

Listing 5 Specification of vault/state-idle-req-inter in CVL

```
invariant state_idle_req_inter()
 currentContract.state == Vault.States.IDLE || currentContract.state == Vault.States.REQ;
```

Listing 6 Specification of vault/state-idle-req-inter in MSL as struct invariant

```
spec vault_addr::vault { spec Vault { invariant (state == IDLE) || (state == REQ); } }
```

Single-transition invariants. We denote by "single-transition invariants" properties of the form "*for every reachable state* s*, and for every transaction* T*, either* T *aborts, or it holds that* $\mathrm{P}(\mathrm{s}, \mathrm{next}(\mathrm{s}, \mathrm{T}), \mathrm{T})$", where $\mathrm{next}(\mathrm{s}, \mathrm{T})$ is the state after a successful execution of T in s. Note that function specs are a special case where the called function is fixed. Certora is quite flexible for the verification of such properties, and allows to express arbitrary (quantifier-free) conditions on the parameters of T. In the Move Prover, there are two different ways to express single-transition invariants, both of which are less general than Certora rules. The first way is to use *global invariant updates*. This construct, however, does not allow to make explicit mention of the parameters of the transaction T, restricting expressible properties to those of the form $\mathrm{P}(\mathrm{s}, \mathrm{next}(\mathrm{s}, \mathrm{T}))$, where T remains implicitly universally quantified. The second way is to use a *schema* of function specs (that is, syntactic sugar to group together a set of function specs with a common body). Writing a single-transition invariant this way, however, requires to write an instance of the schema for each method, making the MSL spec significantly more verbose than in CVL. As an example, consider the property:

bank/assets-dec-onlyif-deposit: "*if the assets of a user* A *are decreased after a transaction, then that transaction must be a* `deposit()` *where* A *is the sender*"

In CVL, it is possible to succinctly express such property as follows:

Listing 7 Specification of bank/assets-dec-onlyif-deposit in CVL

```
rule assets_dec_onlyif_deposit {
  env e; method f; calldataarg args; address a;
  require e.msg.sender != currentContract && a != currentContract;

  mathint old_a_balance = nativeBalances[a];
  f(e, args); // non-reverting call to an arbitrary function f of the Bank contract
  mathint new_a_balance = nativeBalances[a];

  assert new_a_bal < old_a_bal =>  // if the balance has decreased...
    (f.selector == sig:deposit().selector && e.msg.sender == a); // ...then f=deposit
}
```

In MSL, it is only be possible to specify the contrapositive, i.e. that, for every transaction that is not a `deposit()`, or for which A is not the sender, then the assets of A are not decreased. This, however, requires to write a spec for each function except `deposit()`, and one further function spec for the `deposit()`, restricted to the case of A not being the sender.

Listing 8 Specification of bank/assets-dec-onlyif-deposit in MSL

```
spec bank_addr::bank {
  spec withdraw {
    let a = signer::address_of(sender);
    let old_a_bal = global<coin::CoinStore<AptosCoin>>(a).coin.value;
    let post new_a_bal = global<coin::CoinStore<AptosCoin>>(a).coin.value;
    requires !features::spec_is_enabled(features::COIN_TO_FUNGIBLE_ASSET_MIGRATION);
    ensures new_a_bal >= old_a_bal;
  }
  spec deposit {
    let a = signer::address_of(sender);
    ensures forall b: address where b!=a : // b is not the sender
              global<coin::CoinStore<AptosCoin>>(b).coin.value
      >= old(global<coin::CoinStore<AptosCoin>>(b).coin.value);
  }
}
```

Note that, in the case bank had a greater number of functions, the size of the MSL specification would grow proportionally, while the CVL spec size would remain constant.

Multiple transition invariants. We denote by "multiple-transition invariants" properties of the form "*for every reachable state* s*, and for every sequence of transactions* $\vec{T} = \mathrm{T}_1 \dots \mathrm{T}_n$*, either one transaction aborts, or* $\mathrm{P}(\mathrm{s}, \mathrm{next}(\mathrm{s}, \vec{T}[1{:}1]), \dots, \mathrm{next}(\mathrm{s}, \vec{T}[1{:}n]), \mathrm{T}_1 \dots \mathrm{T}_n)$ holds", where $\mathrm{next}(\mathrm{s}, \vec{T}[1{:}i])$ denotes the state after the successful execution of $\mathrm{T}_1, \dots, \mathrm{T}_i$. In CVL, it is possible to express such specifications analogously to single-transition invariants, by subsequent function calls in the same rule. In MSL, this kind of specifications does not seem to be expressible. For example, consider the property:

vault/finalize-or-cancel-twice-revert: "*a* `finalize()` *or a* `cancel()` *transaction aborts if performed immediately after another* `finalize()` *or* `cancel()` *transaction.*"

This is not expressible in MSL, while Certora can verify the following CVL spec:

Listing 9 Specification of vault/finalize-or-cancel-twice-revert in CVL

```
rule finalize_or_cancel_twice_revert {
    env e1, e2; bool b1, b2; // environments and selectors for transactions tx1,tx2
    if (b1) { finalize(e1); } else { cancel(e1); } // tx1 performs finalize of cancel
    if (b2) { finalize@withrevert(e2); } else { cancel@withrevert(e2); } // same for tx2
    assert lastReverted; // checks that the 2nd tx is always reverted
}
```

Metamorphic properties. These are properties that involve multiple finite sequences of transactions [14]. A typical class of metamorphic property are *additivity properties*: e.g.:

bank/deposit-additivity: "*two successful* `deposit()` *of* n_1 *and* n_2 *units of token* T *performed by the same sender are equivalent to a single* `deposit()` *of* $n_1 + n_2$ *units of* T"

In CVL, it is possible to express some metamorphic properties through the use of `storage` types, which allow to record the contract storage at different points of execution and to later compare them (see, e.g. Listing 10). This feature is not present in MSL, so metamorphic properties do not seem expressible.

Listing 10 Specification of bank/deposit-additivity in CVL (simplified)

```
using Bank as c;
rule withdraw_additivity {
    env  e1, e2, e3; // environments for transactions tx1,tx2,tx3
    uint v1, v2, v3; // values sent along with transactions tx1,tx2,tx3
    storage initial = lastStorage; // save the current storage in variable initial

    require e1.msg.sender == e2.msg.sender; // the senders of tx1,tx2 must be equal
    require v1+v2 <= currentContract.opLimit;
    withdraw(e1,v1); withdraw(e2,v2); // perform tx1,tx2 in sequence
    storage s12 = lastStorage; // saves the current storage in variable s12

    require e3.msg.sender == e1.msg.sender; // the sender of tx3 is the same as tx1,tx2
    require v3 == v1+v2; // the amount of tx3 must be the sum of the amounts in tx1,tx2
    withdraw(e3,v3) at initial;
    storage s3 = lastStorage; // saves the current storage in variable s3

    assert s12[c] == s3[c];   // checks that tx1;tx2 have the same effect of tx3
}
```

Other properties. Some classes of properties do not seem expressible in any of the two tools. Without claiming exhaustivity, we now briefly discuss some of the classes we have encountered, with particular attention to those that seem addressable by other tools.

Liveness. Liveness properties have the form "*eventually a state that satisfies certain conditions is reached*". In price-bet, a desirable liveness property is

price-bet/eventually-balance-zero: "*eventually the contract balance goes to* 0"

Note that this property is closely related to, but more abstract than, the property

price-bet/timeout-not-revert: "*a transaction* `timeout()` *[which transfers the assets controlled by the contract to the owner] does not revert if the deadline has passed*"

Tools able to handle such kind of properties, usually under the assumption of fairness conditions (in the example, that the `timeout()` function is called at least once after the deadline), are VeriSolid [19], VeriMove [21], and SmartPulse [31].

Liquidity/Enabledness. Liquidity [7] or Enabledness [29] properties are of the form "*in every reachable state, certain users are always able to fire a (fixed) number of transactions to reach a desirable state*". In bank, an example of such properties is

bank/no-frozen-credits: "*if the credits are strictly positive, it is possible to reduce them*"

Note that this kind of property never mentions the function that should be called nor its parameters, as they are existentially quantified and determining them (as a function of the current state) is a task of the tool. A tool that addresses such kind of properties is Solvent [7].

CTL fragment: The specification language of VeriSolid (and, consequently, of VeriMove) covers an expressive fragment of Computational Tree Logic (CTL). Such expressivity comes at the expenses of soundness, as the verification process relies on a certain level of abstraction. Examples of CTL specifications include the Liveness seen before, as well as properties of the form "P_1 *cannot happen after* P_2", or "*If* P_1 *happens, then* P_2 *can only happen after* P_3 *happens*". These properties cannot be expressed in CVL, since it is not possible to talk about unbounded sequences of method calls, but only about sequences of states of finite length. E.g., a property not expressible in CVL but in the CTL fragment supported by VeriSolid is:

vault/finalize-after-withdraw-not-revert: "*after a successful* `withdraw()`*, if no* `cancel()` *or* `finalize()` *have been called successfully, then* `finalize()` *does not abort*"

All these properties have a higher level of abstraction than those discussed in the previous paragraphs. Although some of these properties, in certain cases, can be reformulated in terms of more concrete properties that imply them, doing so requires a more advanced knowledge of the low-level aspects, and reduces their generality. It has been observed that properties that abstract the system have a better return-on-investment than low-level properties [38].

Orthogonal features of properties. We finally address specific features of properties that can appear in all previous classes, hence for which a separate discussion is needed.

Inter vs. Intra function invariants Invariants can be of two kinds: those that must be preserved across function calls (*inter-function* invariants) and those that must be preserved within the execution of a function (*intra-function* invariants). In the latter, the notion of *reachable state* is extended to intermediate states. In some cases, intra-function invariants give stronger security guarantees. For example, consider the invariant

`vault/keys-invariant-inter`: "*the* `receiver` *key cannot be changed after initialization*"

Requiring this invariant to only hold inter-function is not enough, as it does not capture attacks where an adversary (i) changes the `receiver` key within `finalize()` before the transfer, (ii) sends the contract tokens to her address, and (iii) restores the key to the original value before the end of the function. It is necessary to require the invariant to hold also intra-function (`vault/keys-invariant-intra`). In Certora, verification of intra-function invariants is possible through ghost variables and hooks [10]. In Move, on the contrary, verification of intra-function invariants is, in general, not possible. The MVP can check that an invariant holds *globally*, i.e. every time the global state is updated [4], but this cannot capture every change that occurs during the execution of a function. In the example attack mentioned above, the MVP is not able to detect that that the `receiver` key is changed within the execution of the `finalize()`.

Nested quantifiers. Several interesting properties require the nesting of quantifiers. In Certora, quantifier nesting is limited to *exists-forall* fragments, whereas *forall-exists* fragments are disallowed. This makes not expressible in CVL properties such as:

`bank/exists-at-least-one-credit-change`: "*after a successful transaction, the credits of at least one account have changed*"

Although MSL allows arbitrary combinations of quantifiers, in practice the verification of such properties can be problematic, as the underlying SMT solvers often struggle with quantifiers. In our experiments, we managed to successfully verify the previous property, but got an inconsistent result in the case of `bank/exists-unique-asset-change`. This inconsistency may be caused by the version of the underlying SMT solver used.

Gas. As discussed in "*Assets transfer*" in Section 4.1, the truth of certain properties may depend on the amount of gas available to the involved functions. For instance, in Solidity the ground truths of `bank/withdraw-assets-transfer` and `vault/finalize-assets-transfer` differ because of the functions used in the respective contracts to transfer ETH from the contract to another address: in the implementation of `bank`, we are using `transfer`, which do not carry enough gas to perform further calls, while in `vault` we are using `call`, which instead transfers all the gas to the callee. Certora however over-approximates the amount of gas available, so it gives a false negative for `bank/withdraw-assets-transfer`.

5 Conclusions

The empirical analysis of our study validates the folklore knowledge that Move is better suited for verification than Solidity. In particular, Move's resource-orientation facilitates the verification of properties concerning, e.g., resource preservation, ownership, and transferring of assets. The only weak spot we have observed in (Aptos) Move is the lack of a construct to enforce the immutability of contract variables – a feature that instead is present in Solidity. We have noted that, in order to properly specify certain properties and determine their truth, some low-level aspects of the underlying contract layers must be taken into account. While this could be discouraging for smart contract developers unfamiliar with these low-level details, it can also serve as an incentive to deepen their understanding on these aspects, ultimately leading to more secure smart contract implementations.

Concerning verification tools, we have observed that the Certora Prover can express a broader set of properties than the Move Prover, e.g., transition invariants involving multiple transactions, metamorphic properties, and intra-function invariants. We believe that all the functionalities needed to verify such properties could be smoothly added to the Move Prover, as well. We have also noted that there are several relevant classes of properties that are out of the scope of both tools (e.g., liveness, liquidity/enabledness, and, more generally, other complex temporal properties concerning the business logic of the contract). We have observed that some of these properties can be addressed by other tools, although their current maturity level remains below that of the Certora and Move provers.

We have contributed with an open dataset of smart contract implementations and verification tasks performed in the two tools (the first of this kind), that we envision will further encourage research on formal verification of Solidity and Move.

Limitations. Although our empirical analysis is based on a set of 66 verification tasks covering a broad range of properties, we expect that extending our dataset would highlight additional differences between verification in Solidity and Move. Moreover, it could reveal some further kinds of properties that would be desirable to verify on real-world smart contracts but currently fall beyond the scope of existing verification tools. This could be the case, e.g., of economic properties of DeFi protocols, whose verification currently requires either using weaker analysis techniques than formal verification (e.g., property-based testing [20], statistical model checking [6]), taint analysis [35, 17], type systems [39, 40], attack synthesis [36] or abstracting from actual contract code [33, 32, 5, 23, 28].

References

1 Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. Solcmc: Solidity compiler's model checker. In *Computer Aided Verification (CAV)*, volume 13371 of *LNCS*, pages 325–338. Springer, 2022. doi:10.1007/978-3-031-13185-1_16.

2 Aptos. Aptos fungible asset (FA) standard. https://aptos.dev/en/build/smart-contracts/fungible-asset, 2025.

3 Aptos. Move security guidelines. https://aptos.dev/en/build/smart-contracts/move-security-guidelines, 2025.

4 Aptos. Move specification language. https://aptos.dev/en/build/smart-contracts/prover/spec-lang, 2025.

5 Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *IEEE Symposium on Security and Privacy (SP)*, pages 2499–2516. IEEE Computer Society, 2023. doi:10.1109/SP46215.2023.10179346.

6 Massimo Bartoletti, James Hsin-yu Chiang, Tommi A. Junttila, Alberto Lluch-Lafuente, Massimiliano Mirelli, and Andrea Vandin. Formal analysis of Lending Pools in Decentralized Finance. In *Int. Symp. on Leveraging Applications of Formal Methods (ISoLA)*, volume 13703 of *LNCS*, pages 335–355. Springer, 2022. doi:10.1007/978-3-031-19759-8_21.

7 Massimo Bartoletti, Angelo Ferrando, Enrico Lipparini, and Vadim Malvone. Solvent: Liquidity verification of smart contracts. In *Integrated Formal Methods (iFM)*, pages 256–266. Springer-Verlag, 2024. doi:10.1007/978-3-031-76554-4_14.

8 Massimo Bartoletti, Fabio Fioravanti, Giulia Matricardi, Roberto Pettinau, and Franco Sainas. Towards benchmarking of Solidity verification tools. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASIcs*, pages 6:1–6:15, 2024. doi:10.4230/OASICS.FMBC.2024.6.

9 Certora. The Certora Verification Language. https://docs.certora.com/en/latest/docs/cvl/index.html, 2025.

10 Certora. Hooks. https://docs.certora.com/en/latest/docs/cvl/hooks.html, 2025.

11 Certora. Reports. https://www.certora.com/reports, 2025.

12 Certora. Working with multiple contracts. https://docs.certora.com/en/latest/docs/user-guide/multicontract/index.html, 2025.

13 Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Ben Livshits. Smart contract and DeFi security: Insights from tool evaluations and practitioner surveys. In *International Conference on Software Engineering (ICSE)*, pages 60:1–60:13. ACM, 2024. doi:10.1145/3597503.3623302.

14 Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):4:1–4:27, 2018. doi:10.1145/3143561.

15 Daniel Jackson, Chandrakana Nandi, and Mooly Sagiv. Certora technology white paper. https://docs.certora.com/en/latest/docs/whitepaper/index.html, 2022.

16 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf.

17 Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. DeFiTainter: Detecting price manipulation vulnerabilities in DeFi protocols. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1144–1156, 2023. doi:10.1145/3597926.3598124.

18 Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022. doi:10.1109/ACCESS.2022.3169902.

19 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-design smart contracts for Ethereum. In *Financial Cryptography and Data Security*, pages 446–465. Springer, 2019. doi:10.1007/978-3-030-32101-7_27.

20 Mikkel Milo, Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Finding smart contract vulnerabilities with concert's property-based testing framework. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 105 of *OASIcs*, pages 2:1–2:13, 2022. doi:10.4230/OASICS.FMBC.2022.2.

21 Keerthi Nelaturu, Eric Keilty, and Andreas Veneris. Natural language-based model-checking framework for Move smart contracts. In *Software Defined Systems (SDS)*, pages 89–94, 2023. doi:10.1109/SDS59856.2023.10328964.

22 Keerthi Nelaturu, Anastasia Mavridou, Emmanouela Stachtiari, Andreas G. Veneris, and Aron Laszka. Correct-by-design interacting smart contracts and a systematic approach for verifying ERC20 and ERC721 contracts with VeriSolid. *IEEE Trans. Dependable Secur. Comput.*, 20(4):3110–3127, 2023. doi:10.1109/TDSC.2022.3200840.

23 Eske Hoy Nielsen, Danil Annenkov, and Bas Spitters. Formalising decentralised exchanges in Coq. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 290–302. ACM, 2023. doi:10.1145/3573105.3575685.

24 OpenZeppelin. The Parity Wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7, 2017.

25 OpenZeppelin. Utilities / address. https://docs.openzeppelin.com/contracts/4.x/api/utils#Address, 2024.

26 Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen. Securing Aptos framework with formal verification. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASIcs*, pages 9:1–9:16, 2024. doi:10.4230/OASICS.FMBC.2024.9.

27 Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. VerX: Safety verification of smart contracts. In *IEEE Symposium on Security and Privacy*, pages 1661–1677. IEEE, 2020. doi:10.1109/SP40000.2020.00024.

28 Daniele Pusceddu and Massimo Bartoletti. Formalizing Automated Market Makers in the Lean 4 Theorem Prover. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASIcs*, pages 5:1–5:13, 2024. doi:10.4230/OASICS.FMBC.2024.5.

29 Jonas Schiffl and Bernhard Beckert. A practical notion of liveness in smart contract applications. In *International Workshop on Formal Methods for Blockchains (FMBC)*, volume 118 of *OASIcs*, pages 8:1–8:13, 2024. doi:10.4230/OASICS.FMBC.2024.8.

30 Shuwei Song, Jiachi Chen, Ting Chen, Xiapu Luo, Teng Li, Wenwu Yang, Leqing Wang, Weijie Zhang, Feng Luo, Zheyuan He, Yi Lu, and Pan Li. Empirical study of Move smart contract security: Introducing MoveScan for enhanced analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1682–1694. ACM, 2024. doi:10.1145/3650212.3680391.

31 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. SmartPulse: Automated checking of temporal properties in smart contracts. In *IEEE Symposium on Security and Privacy (SP)*, pages 555–571. IEEE, 2021. doi:10.1109/SP40001.2021.00085.

32 Xinyuan Sun, Shaokai Lin, Vilhelm Sjöberg, and Jay Jie. How to exploit a DeFi project. In *Workshop on Trusted Smart Contracts*, volume 12676 of *LNCS*, pages 162–167. Springer, 2021. doi:10.1007/978-3-662-63958-0_14.

33 Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal analysis of composable DeFi protocols. In *Workshop on Trusted Smart Contracts*, volume 12676 of *LNCS*, pages 149–161. Springer, 2021. doi:10.1007/978-3-662-63958-0_13.

34 Palina Tolmach, Yi Li, Shangwei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Comput. Surv.*, 54(7):148:1–148:38, 2022. doi:10.1145/3464421.

35 Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.*, 3(OOPSLA):189:1–189:29, 2019. doi:10.1145/3360615.

36 Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. FORAY: towards effective attack synthesis against deep logical vulnerabilities in defi protocols. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1001–1015. ACM, 2024. doi:10.1145/3658644.3690293.

37 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard J. Trefler, Valentin Wüstholz, and Arie Gurfinkel. Verifying Solidity smart contracts via communication abstraction in SmartACE. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 13182 of *LNCS*, pages 425–449. Springer, 2022. doi:10.1007/978-3-030-94583-1_21.

38 Meng Xu. Research report: Not all Move specifications are created equal : A case study on the formally verified Diem Payment Network. In *Workshop on Language-Theoretic Security (LangSec)*, pages 200–214. IEEE, 2024. doi:10.1109/SPW63631.2024.00024.

39 Siqiu Yao, Haobin Ni, Andrew C. Myers, and Ethan Cecchetti. SCIF: A language for compositional smart contract security. *CoRR*, abs/2407.01204, 2024. doi:10.48550/arXiv.2407.01204.

40 Brian Zhang. Towards finding accounting errors in smart contracts. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 138:1–138:13. ACM, 2024. doi:10.1145/3597503.3639128.

41 Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 615–627. IEEE, 2023. doi:10.1109/ICSE48619.2023.00061.

42 Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. The Move Prover. In *Computer Aided Verification (CAV)*, volume 12224 of *LNCS*, pages 137–150. Springer, 2020. doi:10.1007/978-3-030-53288-8_7.