Rich Specifications for Ethereum Smart Contract Verification

CHRISTIAN BRÄM, ETH Zurich, Switzerland
MARCO EILERS, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland
ROBIN SIERRA, ETH Zurich, Switzerland
ALEXANDER J. SUMMERS, University of British Columbia, Canada

Smart contracts are programs that execute in blockchains such as Ethereum to manipulate digital assets. Since bugs in smart contracts may lead to substantial financial losses, there is considerable interest in formally proving their correctness. However, the specification and verification of smart contracts faces challenges that rarely arise in other application domains. Smart contracts frequently interact with unverified, potentially adversarial outside code, which substantially weakens the assumptions that formal analyses can (soundly) make. Moreover, the core functionality of smart contracts is to manipulate and transfer resources; describing this functionality concisely requires dedicated specification support. Current reasoning techniques do not fully address these challenges, being restricted in their scope or expressiveness (in particular, in the presence of re-entrant calls), and offering limited means of expressing the resource transfers a contract performs.

In this paper, we present a novel specification methodology tailored to the domain of smart contracts. Our specifications and associated reasoning technique are the first to enable: (1) sound and precise reasoning in the presence of unverified code and arbitrary re-entrancy, (2) modular reasoning about collaborating smart contracts, and (3) domain-specific specifications for resources and resource transfers, expressing a contract's behaviour in intuitive and concise ways and excluding typical errors by default. We have implemented our approach in 2Vyper, an SMT-based automated verification tool for Ethereum smart contracts written in Vyper, and demonstrated its effectiveness for verifying strong correctness guarantees for real-world contracts.

CCS Concepts: • **General and reference** → **Verification**; • **Theory of computation** → *Programming logic*; Automated reasoning; *Invariants*; **Program specifications**; **Program verification**; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Ethereum, smart contracts, specification, software verification, resources

ACM Reference Format:
Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich Specifications for Ethereum Smart Contract Verification. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 146 (October 2021), 30 pages. https://doi.org/10.1145/3485523

1 INTRODUCTION

Smart contracts are programs that execute in blockchains such as Ethereum, and allow the execution of resource transactions between different parties without the need for a trusted third party. Smart contracts tend to be comparatively short but non-trivial programs, and since any bugs can and do

Authors' addresses: Christian Bräm, c.braem@gmx.ch, Department of Computer Science, ETH Zurich, Switzerland; Marco Eilers, marco.eilers@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Peter Müller, peter.mueller@inf.ethz.ch, Department of Computer Science, ETH Zurich, Switzerland; Robin Sierra, robin.sierra@outlook.com, Department of Computer Science, ETH Zurich, Switzerland; Alexander J. Summers, alex.summers@ubc.ca, Department of Computer Science, University of British Columbia, Canada.

2475-1421/2021/10-ART146
https://doi.org/10.1145/3485523

frequently lead to the loss of potentially-large amounts of money [Güçlütürk 2018], using formal methods to ensure their correctness is both practically viable and highly desirable.

Compared to other programs, smart contracts bring reasoning challenges that are insufficiently supported by classical verification techniques: First, smart contracts usually call other smart contracts, for instance, to perform transactions. These other contracts are typically developed by unknown parties and cannot be assumed to be verified; they might even exhibit adversarial behaviour to gain a financial advantage. As a result, standard modular reasoning techniques such as separation logic [Reynolds 2002], which reason about calls under the assumption that *all* code is verified, do not apply in this setting. This problem is exacerbated by the presence of *re-entrant* calls, i.e. callbacks from functions called by the contract itself. A sound reasoning technique must account for *all* behaviours a call to an unverified contract could possibly exhibit, which requires novel ways of specifying the calling contract. Second, even in this adversarial setting, multiple smart contracts may form a distributed application designed to collaboratively maintain invariants across the contracts' states. Such collaborations are often decoupled using interface abstractions, which must be equipped with sufficient specification to soundly guarantee the preservation of these invariants. Third, typical smart contracts are primarily concerned with modelling and executing *resource transactions* of different kinds, ranging from token contracts to escrow implementations, ICOs, and complex decentralised finance (DeFi) applications. However, even though notions such as resource ownership and agreed exchanges are central to programmer intentions, resources themselves are often implicit in smart contract implementations. This discrepancy between high-level intentions and low-level implementations easily leads to subtle and potentially-costly mistakes.

Existing (automated) verifiers for smart contracts do not fully address these challenges. Some verifiers [Feist et al. 2019; Lai and Luo 2020; Tikhomirov et al. 2018; Tsankov et al. 2018] prove specific properties, e.g. the absence of overflows or re-entrancy bugs, but cannot be used to prove full functional correctness of a contract. Other verifiers [Hajdu and Jovanovic 2019; Hildenbrandt et al. 2018; Kalra et al. 2018; Permenev et al. 2020] aim to prove arbitrary, user-defined properties using variations of established specification and verification techniques. However, because these techniques are not sufficiently adapted to the setting of smart contracts, they are either not generally applicable or very imprecise in the presence of arbitrary re-entrancy. In general, no existing technique allows reasoning modularly about compositions of multiple smart contracts while preserving interface abstractions. Furthermore, existing techniques offer limited support for specification and reasoning in terms of high-level notions of custom resources such as tokens.

In this paper, we propose a novel specification and verification methodology for the sound, unbounded verification of general (safety) properties of Ethereum smart contracts. We offer specification constructs tailored to the domain of smart contracts, enabling users to prove strong functional correctness properties of arbitrary smart contracts, with specifications that capture their intended resource manipulations explicitly. We make the following four main contributions:

(1) Reasoning in the presence of unverified code. To the best of our knowledge, we present the first smart-contract verification technique that is sound and precise in the presence of calls to unverified contracts with arbitrary re-entrancy. Our technique can prove that properties cannot be invalidated by calls to unverified code, including vital properties such as access control to resources.

(2) Modular reasoning about collaborating smart contracts. We explain the challenges of verifying collaborating contracts; our specification methodology can capture all required information at the interface level, providing the first *modular* verification technique for collaborating smart contracts.

(3) Native resource-oriented specifications. We introduce novel specification features for directly capturing resource ownership, transfer, offers to exchange, and loaning, enabling direct and concise specification of programmer intentions. Ubiquitous properties of resources such as ownership,

```
beneficiary: address
highestBid: int256
highestBidder: address
ended: bool
pendingReturns: map(address, int256)
lock: bool

def bid():
  assert not self.lock and msg.value > self.highestBid and not self.ended
  self.pendingReturns[self.highestBidder] += self.highestBid
  self.highestBidder = msg.sender
  self.highestBid = msg.value

def withdraw():
  assert not self.lock
  toSend = self.pendingReturns[msg.sender]
  self.pendingReturns[msg.sender] = 0
  self.lock = True
  send(msg.sender, value=toSend)
  self.lock = False

def end():
  assert not self.lock and not self.ended and msg.sender == self.beneficiary
  self.ended = True
  self.lock = True
  send(self.beneficiary, value=self.highestBid)
  self.lock = False
  self.highestBid = 0
```

Fig. 1. Simplified auction contract written in Vyper. **assert** commands revert the current transaction, whereas send commands send Ether to another contract. Since the contract does not have an explicit constructor function, all fields are initialized with default values.

access control, and non-duplicability are baked into our system, avoiding potentially repetitive and error-prone boilerplate specifications; violations of these properties are found by default.

(4) Implementation and evaluation. We implemented our approach in 2Vyper, an automated, SMT-based verification tool for the Vyper language [Ethereum 2021c] for Ethereum smart contracts. It supports the entire current Vyper language and allows specifying contracts and interfaces in the form of readable, source-level code annotations. Our evaluation shows that 2Vyper enables automated verification of strong correctness properties of (collaborating) real-world contracts with reasonable performance and annotation overhead. In particular, we demonstrate that 2Vyper can verify contracts that use re-entrancy patterns unsupported by other verification tools, and that it enables modular verification of collaborating smart contracts used in practice. Our implementation and evaluation are available as an artifact [Bräm et al. 2021].

Outline. The paper is structured as follows: We introduce Ethereum smart contracts in Sec. 2. In Sec. 3, we informally introduce the specification constructs we use to reason about contracts containing re-entrant calls; subsequently, we show how they can be used to reason about collaborating contracts in Sec. 4. We introduce our resource-based specification approach in Sec. 5, and present our verification technique in the form of a Hoare logic in Sec. 6. We describe our implementation in 2Vyper and evaluate it in Sec. 7. We discuss related work in Sec. 8 and conclude in Sec. 9.

2 ETHEREUM SMART CONTRACTS

Ethereum smart contracts are programs usually written in a high-level language, most-commonly Solidity [Ethereum 2021b] or the newer Vyper [Ethereum 2021c] language, and then compiled to bytecode for execution in the Ethereum Virtual Machine (EVM) [Wood et al. 2014]. Fig. 1 shows an example of a Vyper smart contract implementing an auction. Note that in this example and throughout the paper, we use a simplified presentation of Vyper and Ethereum contracts and omit details that are irrelevant to our approach (e.g. that Ether can be transferred only by calling functions marked as payable, or that Vyper functions revert when encountering under- or overflows[1]). We also ignore the fact that contract execution consumes *gas*, i.e. a fixed cost associated with every executed instruction, which is not relevant for proving safety properties, the focus of this paper.

A contract can declare *fields* that form its persistent state. In our example, the contract stores the beneficiary of the auction, the current highest bid and bidder, and the amounts of *wei*, a sub-unit of Ethereum's built-in currency *Ether*, it owes to bidders who have been outbid. In addition to explicitly declared fields, every contract has a built-in balance field that tracks the amount of Ether currently held by the contract. Unlike ordinary fields, which can be written to directly by the contract (but, crucially, *not* by other contracts), the balance cannot be written to directly. Ether (and wei) is the only resource with native language support; programmers can, however, implement smart contracts that provide custom resources (often called *tokens*), illustrated later in this section.

Contracts define a set of functions and a special constructor function called __init__ that is executed when the contract is set up. Smart contracts are executed as *transactions*: a caller outside the blockchain can request to invoke a contract's function, and miners can then decide to execute this function as part of the next block. If this happens, the function is executed, and may in turn call functions of the same or other contracts as part of the same transaction[2]. (Note that throughout this paper, we inline internal calls to private functions for simplicity.) External calls typically occur via *interfaces* that list (a subset of) the available functions of a contract. Importantly, there is no observable concurrency while all transitively-called functions are executed.

The intended workflow of the auction contract is that clients call the bid function and transfer along a larger amount of Ether than the current highest bid. If another client bids a higher value later, the contract updates pendingReturns to remember that no-longer-highest bidders can get their Ether back. Such a bidder can call withdraw to have the Ether transferred back to them. Contracts can transfer Ether to other contracts via send commands (e.g. in function end), where the parameter value specifies the transferred amount, or by calling a function (like the bid function) on another contract and implicitly passing along some amount of Ether. Internally, these are the same: executing a send command is implemented by calling a default function on the recipient.

Ethereum transactions can *revert*, meaning they abort and all state changes they made are reset, for several reasons. Smart contracts commonly use **assert** commands to revert a transaction if the asserted condition is false. This is intentionally-possible behaviour used to enforce that e.g. arguments supplied to a call are valid and that the call is allowed given the current state of the called contract. For example, a call to the end function will revert if the auction is already over, and bid reverts if the new bid is not higher than the current highest bid. This contract also reverts if called while the lock field is set, a pattern commonly used to explicitly prevent a contract from being called in unexpected situations (often to prevent re-entrancy vulnerabilities, discussed below).

In addition to the contract's fields and explicitly declared arguments, a contract can always access the implicit arguments msg and block, which contain information about the current call

[1]Our tool nonetheless allows one to verify that a function does not revert due to under- or overflows.
[2]Throughout this paper, when we say that a contract interacts with other contracts, we mean other contract *instances*, i.e. contracts deployed at other addresses that may contain the same or (usually) different code from our contract.

```
1   minter: address
2   balances: map(address, int256)
3
4   def transfer(from: address, to: address, amount: uint256):
5     assert self.balances[from] >= amount and msg.sender == from
6     newAmount: int256 = self.balances[from] - amount
7     self.balances[to] += amount
8     self.balances[from] = newAmount
9     to.notify(from, self, amount)
10
11  def mint(to: address, amount: uint256):
12    assert msg.sender == self.minter
13    self.balances[to] += amount
```

Fig. 2. Simplified token contract implemented in Vyper. The minter can create new tokens by calling mint; other users call transfer to give their own tokens to another user. **assert** statements ensure that the transaction reverts if a user tries to spend tokens they do not own.

and the block the current transaction is a part of. For example, msg has the particularly important field msg.sender, which contains the address of the caller of the current function. Function end uses this variable to ensure that only the beneficiary of the auction can end the auction, whereas the bid function uses msg.value to obtain the amount of Ether sent with the call.

Custom resources. While the auction contract works directly with the built-in Ether currency, many real contracts implement or work with *tokens* [Vogelsteller and Buterin 2015], i.e. custom currencies tracked via ad-hoc implementations in smart contract fields. Fig. 2 shows a very simple version of a token contract. Its state consists of a map that represents the balances that each other contract holds for this token. Contracts can call transfer to transfer tokens from one contract to another, which simply corresponds to updating the map. This contract enforces important properties common to resources in general: Each client holding a balance should *be able to transfer only tokens that it owns.* This implicit notion of resource *ownership* (tracked via numeric values in a map, here) is a native notion in our specification methodology, explained in Sec. 5. This contract's implementation enforces this intention by reverting if it is asked to transfer tokens away from anyone except the caller. Similarly, the right to mint *new* tokens is restricted to a special privileged contract (represented by its *address*), self.minter.

The token contract can also be used to illustrate the infamous concept of *re-entrancy vulnerabilities*: the subtle potential for a called contract to perform malicious callbacks and achieve undesirable outcomes. Say, for example, that lines 8 and 9 in the token contract were swapped, i.e. the contract first called the receiver contract to notify it that it has received tokens, *before* reducing their balance. If the notified contract called the sender of the transaction, it could in turn call back into the token contract and transfer the tokens it just transferred away *a second time*; in particular, the **assert** on line 5 would not prevent the transfer because the balance was not yet updated at the time the callback happens. This would allow clients of the token contract to create tokens out of thin air. Variations of this pattern are behind most re-entrancy vulnerabilities, e.g. the infamous DAO exploit [Güçlütürk 2018]; as we will show in Sec. 5, our explicit resource reasoning will uncover such coding errors by default.

3 VERIFICATION IN THE PRESENCE OF UNVERIFIED CODE AND RE-ENTRANCY

Smart contracts frequently interact with other contracts; in particular, contracts that offer services to arbitrary clients often call functions on arbitrary other contracts. For example, the auction contract above sends Ether to (i.e. calls) an arbitrary msg.sender in its withdraw function, which is

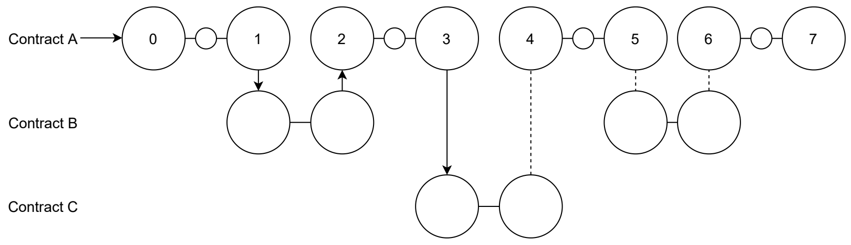

Fig. 3. Example of a smart contract transaction involving re-entrancy. Calls and returns are marked by arrows and dashed lines, respectively.

necessary to ensure that any contract that has previously placed a bid and has been outbid can get back the Ether they sent. While calls to functions of the same contract (*internal* calls) can simply be verified by inlining the callee function, calls to other contracts (*external* calls) are challenging for two reasons. First, as we explained earlier, the implementations of other contracts are in general not verified, and so we cannot reason about external calls using e.g. standard pre- and postconditions. Second, the implementations of other contracts are in general not known: we cannot make any assumptions about the callbacks they perform directly or via other contracts[3]. That is, we do not know if an external call simply modifies the state of the called contract and then returns, or if it triggers more complicated interactions such as those shown in Fig. 3: Contract A calls contract B, which performs a re-entrant call to A; subsequently, A performs an external call to a third contract C before all calls return. In this scenario, when A's call to B returns, its own state may have changed as a result of the re-entrant call from B. Consequently, if the implementation of B is unknown, one does not know which functions of A have been re-entrantly called (if any), and how A's state has changed as a result. In this section, we present a specification and verification technique that is sound in the presence of unverified code and arbitrary re-entrancy. In the following, we assume that all calls are external (internal calls are inlined) and that the implementation of the callee is neither known nor verified (known or verified callees could provide stronger assumptions about the effects of the calls, but we focus on the common, most difficult case here).

3.1 The Challenge

The problem of re-entrancy by itself is not specific to smart contracts; it can also occur, for example, between objects in object-oriented programming languages. When reasoning about programs in such languages, this problem is usually solved by constraining what a called function may assume about the state and which parts of the heap it may manipulate [Barnett et al. 2004; Kassios 2006; Müller 2002; Reynolds 2002]. However, these verification techniques require that *all* executed code is verified (and therefore known to adhere to the rules of the verification technique). In particular, called functions are typically verified to *only* cause side effects allowed by the verification technique. In a smart contract setting, however, external code is often unverified, potentially malicious, and cannot be soundly assumed to follow *any* particular rules beyond those of the execution environment. For the same reason, classical *preconditions* on public functions are of limited use in this setting, since one cannot rely on external callers actually respecting them. In order to reason soundly, we have to conservatively assume that any external call may lead to arbitrary callbacks into the original contract and, in particular, mutate the original contract's state in any possible way.

[3]It is common to limit the amount of gas sent with a call so that it is not sufficient to perform callbacks, but relying on this is considered bad practice, since the gas cost of Ethereum Virtual Machine instructions can change.

Some existing reasoning techniques for smart contracts either assume [Permenev et al. 2020] or aim to prove [Albert et al. 2020; Grossman et al. 2018] that re-entrancy cannot lead to behaviours that cannot also occur without re-entrancy (i.e. that contracts are *effectively callback-free* (ECF) [Grossman et al. 2018]). However, these techniques do not apply to the increasing number of contracts that use re-entrant calls as an essential part of their intended workflow (e.g. [Minacori 2020], which we explain Sec. 7). In contrast, our methodology applies to *all* contracts, even if they are not ECF: Its central feature is that it allows users to express and prove critical properties of a contract despite the potential for arbitrary re-entrancy, as we explain next.

3.2 Specification and Verification Technique

We propose to use a two-pronged approach to smart contract specification and verification: (1) We introduce a novel specification construct that lets us specify constraints on how a contract directly manipulates its own state. These constraints can be verified without considering external calls. (2) We introduce two additional specification constructs that allow us to reason about external calls, even when the callee functions are unverified and potentially trigger re-entrant calls. Both ingredients exploit a key feature of smart contracts: *All* contract state is private, i.e. cannot be directly modified by functions in other smart contracts. In particular, all updates to a contract's local state are performed by *some* function from the contract's own code.

This feature provides a key distinction between smart contracts and most standard object-oriented languages; however, there are some non-smart contract languages (e.g. actors in Swift [McCall et al. 2021]) that do offer this feature and can therefore also be verified using our approach. Generally, our technique enables code verification in the presence of re-entrancy and unverified code in any language that offers (1) objects (or contracts) with only object-private state, that is, the state of an object can be modified only by methods of that object, and (2) non-aliasing guarantees for object-private state, that is, there are never any mutable outside references to private object state.

Reasoning about call-free code. Since the state of a contract can be modified only by its own functions, one can express many important properties as constraints on local code, that is, the call-free code segments between (external) calls. For instance, if *all* such code segments of a contract only ever increase the value of a counter, its value will never get smaller, even when external, potentially re-entrant functions are called. We refer to call-free sequences of statements as *local segments*. In a sense, they represent the atomic operations a contract can perform: Outsiders can observe a contract's state *between* local segments, but never in the middle of a segment. In Fig. 3, the local segments of contract A are the ones between the state pairs (0, 1), (2, 3), (4, 5) and (6, 7).

One class of properties that can be enforced by imposing constraints only on local segments is access control, i.e. restricting the right to perform certain operations or modify certain data (indirectly, by calling a function on the contract) to specific callers. Access control is particularly important for smart contracts, since, unlike in standard object-oriented programs, they have to store *all* the data of their clients in their own storage (e.g. the balances in the token contract), making it vital to enforce that clients can only modify parts of that storage that conceptually belong to them. Access control restrictions are therefore a necessary part of the public specification of a contract. For example, for the auction contract from Fig. 1 we may want to prove that only the beneficiary can end the auction (by setting the ended flag).

To express constraints on local segments, we introduce *segment constraints*—two-state assertions (i.e. assertions that can refer both to the current state and an **old** state) on the local state of a contract that must hold between the start and end states of *each* local segment in the contract. Segment constraints are specified per contract, not per individual segment. In our auction example, we can express the access restriction for the ended field using the segment constraint msg.sender ≠

```
def end():
  { msg.sender ≠ self.beneficiary ∧ self.ended = x₁ }
  assert not self.lock and not self.ended and msg.sender == self.beneficiary
  self.ended = True
  self.lock = True
  { self.ended = x₁ }
  send(self.beneficiary, value=self.highestBid)
  { msg.sender ≠ self.beneficiary ∧ self.ended = x₂ }
  self.lock = False
  self.highestBid = 0
  { self.ended = x₂ }
```

Fig. 4. Proof obligations generated by segment constraint msg.sender $\neq$ *old*(self . beneficiary) $\Rightarrow$ self .ended = *old*(self .ended) for function end. The function is divided into two local segments by the call in the middle; each segment is call-free and can therefore be verified using standard techniques. We use the logical variables x_1 and x_2 to represent the old values of the ended field. The same property has to be proved for every local segment in all other functions in the contract.

old(self . beneficiary) $\Rightarrow$ self .ended = **old**(self .ended). Since, by definition, there are no external calls between the start and end states of a local segment, segment constraints are verified without considering external, unverified code. Figure 4 illustrates the proof obligations generated by this example constraint for the end function from our auction contract.

Reasoning about (external) calls. An external call may modify the state of the calling contract A only via one or several re-entrant calls. These re-entrant calls perform the modifications of A's state by executing an arbitrary number of A's functions, which in turn will execute some number of A's local segments (in Fig. 3, the segments (2, 3) and (4, 5)). Consequently, the reflexive and transitive closure of constraints describing the effects of A's functions and segments can be used to soundly approximate the effects of an external call. In the following, we introduce two complementary forms of such transitive constraints, which are useful for expressing different kinds of common properties. Both are *auxiliary* specifications in the sense that they do not directly express vital correctness properties (unlike e.g. segment constraints prescribing access control properties), but instead allow us to preserve properties across external calls.

Transitive segment constraints. A vital property the end function of the auction contract must fulfill is that, when it returns, the contract's ended flag is set. Proving this requires showing that any re-entrant calls resulting from the send-command do not set the flag to false . We can show that this is the case because no local segment of the contract ever unsets the flag once it has been set, a property which can be expressed as a segment constraint.

The reflexive and transitive closure of all segment constraints of a contract describes the effect of an arbitrary number of local segments. Thus, it is known to hold between the pre-state and the post-state of any external call (i.e. between states 1 and 6 in Fig. 3) and can be used to reason about such calls. Since it is generally not possible to compute the reflexive, transitive closure of our segment constraints automatically, we allow programmers to specify the *transitive segment constraints* of a contract. These are checked to be reflexive and transitive, and are verified to hold across each local segment of the contract. Like segment constraints, transitive segment constraints are two-state assertions on the local state of a contract, similar to history constraints [Liskov and Wing 1993]. Since *any* sequence of local segments is guaranteed to satisfy the transitive segment constraints of a contract, they may soundly be assumed to hold between the pre- and poststate of each external call. Note that transitive segment constraints do *not* subsume ordinary segment

```
def end():
    { $self.ended = x_1$ }
    assert not self.lock and not self.ended and msg.sender == self.beneficiary
    self.ended = True
    self.lock = True
    { $(x_1 \Rightarrow self.ended) \wedge x_3 = self.ended$ }
    send(self.beneficiary, value=self.highestBid)
    { $self.ended = x_2 \wedge (x_3 \Rightarrow self.ended)$ }
    self.lock = False
    self.highestBid = 0
    { $x_2 \Rightarrow self.ended \wedge self.ended$ }
```

Fig. 5. Proof outline showing the use of the transitive segment constraint *old*(self .ended) ⇒ self .ended to prove the postcondition self .ended. We use logical variables x_1 and x_2 to represent the values of the ended field at the beginning of each local segment, and x_3 to represent its value before the send command. The transitive segment constraint must be proved for every local segment of all functions (blue). It can then be assumed to hold between the pre- and post-states of the external call (green), i.e., if self .ended is true before the call, we may assume it is true after the call. This is sufficient to prove the desired postcondition (red).

constraints, since typical segment constraints used for access control (e.g. the restriction on who can end an auction, shown above) are not transitive.

Transitive segment constraints are useful to express constancy properties, such as the fact that the auction's beneficiary never changes, or monotonicity properties like that of the ended field discussed above. The latter can be expressed using the transitive segment constraint **old**(self .ended) ⇒ self .ended; Figure 5 illustrates how this constraint can be used to prove the desired postcondition for function end (even without the lock, which we will discuss below).

Transitive segment constraints subsume single-state *contract invariants*, which are often useful to specify consistency conditions on contract states, which must hold whenever the contract relinquishes control to other contracts (and, thus, its state becomes observable to the environment). The verification of transitive segment constraints implies that single-state contract invariants hold at the end of each local segment, which includes the state before any call as well as the post-state of each function. Consequently, each function may soundly assume such contract invariants to hold in its pre-state, as well as after the return of each external call, analogously to class or object invariants in object-oriented programs [Drossopoulou et al. 2008; Leavens et al. 2008]. For the auction, an important invariant is that its funds suffice to pay all its obligations, which can be written as the transitive segment constraint self .balance ≥ *sum*(self .pendingReturns) + self .highestBid.

Function constraints. It is common that each individual function of a contract *as a whole* satisfies a two-state property, even if some of its local segments do not. Such situations occur for instance if *some* sequences of local segments violate the property, but no function in the contract ever executes such a sequence. The re-entrancy lock in the auction contract is an example: The field self . lock is set to true by withdraw and end before their calls to send, and reset to false afterwards. Since each function of the contract reverts if the locks is set, this pattern ensures that each function of the auction contract leaves the contract state completely unchanged if the lock is set in its pre-state.

However, this property cannot be verified as a transitive segment constraint: Some local segments reset the lock, such that any subsequent state change violates the property. That is, the property does not hold for arbitrary sequences of local segments, but it does hold between the pre-state and the post-state of each contract function. Note that any external call can modify the contract state only by executing these contract functions (via re-entrant calls) from start to finish.

To exploit this fact, we introduce *function constraints*: two-state assertions on the local state of a contract that must hold between the pre- and post-state of every function in the contract. In

```
interface Token:
  balances: map(address, uint256)

  def transfer(from: address, to: address, amount: uint256):
    pass

contract Auction:
  token: Token

  def withdraw():
    assert not self.lock
    toSend = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    self.lock = True
    self.token.transfer(self, msg.sender, toSend)
    self.lock = False

  def distributeExcess():
    excess: int128 = self.token.balances[self] - sum(self.pendingReturns)
    excess -= self.highestBid
    assert excess != 0
    perBidder: int128 = excess / size(self.pendingReturns)
    for bidder in keys(self.pendingReturns):
      self.pendingReturns[bidder] += perBidder
```

Fig. 6. Minimal interface of the token contract in Fig. 2, and part of an adapted auction contract that deals in tokens and additionally has a function that distributes excess tokens among previous bidders.

Fig. 3, this means they have to hold between states 2 and 5 as well as states 0 and 7. Like transitive segment constraints, function constraints are specified per contract; they must be satisfied by *all* of its functions (reflecting that we do not know statically which re-entrant calls are triggered by an external call). Since external calls may trigger the execution of an arbitrary number of contract functions, we require function constraints to be reflexive and transitive. For the lock example, we can express the desired property as the function constraint $\mathbf{old}(\text{self}.\text{lock}) \Rightarrow \text{self} = \mathbf{old}(\text{self})$.

Note that function constraints do *not* subsume transitive segment constraints. For instance, in the special case of single-state assertions, transitive segment constraints (that is, the contract invariants discussed above) are known to hold before each call and may, thus, be assumed in the pre-state of each function, whereas function constraints may not. Neither do transitive segment constraints subsume function constraints, as we illustrated with the lock example above.

With these two specification constructs, we can modularly verify properties in the presence of calls to unverified contracts with arbitrary re-entrancy. In Sec. 5, we will complement these constraints with effect specifications on a contract's resources to obtain even stronger guarantees.

4 INTER-CONTRACT INVARIANTS

Smart contract applications are frequently implemented via multiple contracts which call one another. As in many programming languages, the *interfaces* of the Vyper and Solidity languages are designed to facilitate such collaborations. Interfaces declare that a contract offers *at least* some set of functions and fields[4], but do not give any information about their implementation, or preclude the existence of additional functions in the contract. Therefore, they decouple client contracts (in the software engineering sense) from the concrete implementations of the contracts they build on.

[4]Interfaces actually contain constant functions that guarantee not to modify any state instead of fields, but we model them as fields here to simplify the presentation.

```
contract BadToken implements Token:

  def steal(from: address, amount: uint256):
    assert self.balances[from] >= amount
    self.balances[msg.sender] += amount
    self.balances[from] -= amount

  def transfer(from: address, to: address, amount: uint256):
    assert self.balances[from] >= amount and msg.sender == from
    newAmount: uint256 = self.balances[from] - amount
    self.balances[to] += amount
    self.balances[from] = 0
    thirdparty.notify()
    assert self.balances[from] == 0
    self.balances[from] = newAmount

contract ThirdParty:
  auction: Auction

  def notify():
    auction.distributeExcess()
```

Fig. 7. Possible implementation of the Token interface. The implementing contract may offer additional functions, e.g. in this case, one that allows anyone to steal tokens from any existing account.

For example, an auction contract similar to our example from Figure 1 could instead deal in tokens conforming to e.g. the ERC20 standard interface [Vogelsteller and Buterin 2015]. Fig. 6 shows a minimal interface of the token contract from Fig. 2 as well as (part of) a modified version of the auction contract, where calls to send are replaced by calls to the token contract's transfer function (we will discuss the added function distributeExcess later).

However, our techniques for equipping contracts with invariants and proof obligations to maintain them no-longer suffice for collaborating contracts, since such collaborations naturally give rise to invariants that depend on the state of *other* contracts. For example, the modified auction still needs an invariant that it has sufficient funds (now tokens) to pay its obligations to all participants, which can be expressed in terms of the states of *both* the auction and token contract by: self.highestBid + *sum*(self.pendingReturns) ≥ self.token.balances[self]. In this section, we extend the technique presented in Sec. 3 to such *inter-contract invariants*[5].

An inter-contract invariant has a single *primary* contract (the contract depending directly on the property); any other contracts whose state is mentioned are its *secondary* contracts. In the example, the modified auction contract is the primary contract, since the auction's funds must be sufficient for the auction to function correctly, whereas the token contract can have many (non-auction) clients with different functionality and is not responsible for their correctness. This asymmetry is reflected in the above invariant, where self is the auction contract.

Ensuring non-trivial inter-contract invariants requires that both the primary and all secondary contracts are verified; the state of unverified contracts may change arbitrarily, which precludes the verification of invariants that depend on it. However, all contracts other than the primary and secondary ones may still be unverified, and as before, verified contracts may still call functions of unverified ones. Additionally, we do *not* depend on having access to the implementations of the secondary contracts. These may in particular be hidden behind interfaces.

[5]We focus on single-state invariants here for simplicity only: our technical solution also supports two-state assertions.

4.1 Challenges

Modular verification of inter-contract invariants poses two main challenges:

Challenge 1: Missing encapsulation. The first challenge is that the state that an inter-contract invariant depends on is not fully-encapsulated in the way we have exploited so far: It is now no longer the case that the invariant can only be broken by code of the primary contract; instead, it can be also be broken by the code of a secondary contract, which may not be known.

To illustrate this challenge, consider a scenario in which the token contract has a function steal that lets an arbitrary contract steal another contract's funds, as shown in Figure 7: If this function existed, a third party could call it to steal the tokens of the auction contract; if the auction contract also has any pending returns, this would break our inter-contract invariant. Note that there is nothing the primary contract can do to prevent this; in fact, this can even happen in a transaction that does not involve the primary contract at all. Additionally, one cannot conclude from the token contract's interface alone whether or not it has such a function (or any other function that allows one contract to decrease another contract's funds).

Challenge 2: Temporarily-broken invariants. Second, secondary contracts may *temporarily* break inter-contract invariants when called by the primary contract in a way that makes the inconsistent state visible to other contracts. Note that it is normal and unavoidable that invariants are temporarily broken; however, states in which this is the case must never be visible to outside contracts, which can be the case here. This challenge is illustrated in function transfer in Figure 7. This function performs a token transfer from one contract to another, as it should, but (perhaps as a clumsy attempt to avoid a DAO-like re-entrancy vulnerability) it temporarily sets the balance of the token sender to zero and performs a call to the outside, before restoring the balances to the desired end state. This can lead to problematic behaviour: Assume that the auction contract has non-zero pending returns for two contracts, A and B, and that contract B is the ThirdParty contract shown in Figure 7. If contract A calls withdraw, the auction contract will call transfer, which will set its token balance to zero. Now the token contract calls function notify of contract B. Note that, in this state, the inter-contract invariant is broken: The auction contract still has pending returns for B, but its current balance is zero. When B in turn calls the function distributeExcess in the auction contract this state, this function does not work as designed: The purpose of the function is to distribute any excess tokens the auction contract may own among previous bidders (which may be part of some intended functionality where third parties are rewarded for taking part in the auction, or simply a failsafe in case someone accidentally transfers tokens to the auction contract). It calculates the excess by subtracting the sum of the pending returns and the current highest bid from the auction contract's token balance, assuming that the result will be non-negative, which *should* be guaranteed by the invariant. As a result, since we assume the invariants (transitive segment constraints) at the beginning of each function, we could prove a postcondition here that states that pending returns can only be increased by this function. Now, however, the result can actually be negative, and as a result, the pending returns of all previous bidders will be decreased, breaking the postcondition. We must therefore adjust our verification technique to ensure that this invariant cannot be proved for this contract, since it does not hold in practice.

Note that, again, it is not possible to see from the interface that this problem exists: If function transfer has a postcondition that describes its behaviour, it will state that (by the time the function returns) the transfer has been executed as desired; nor is it possible to see from an interface whether the function performs any calls to the outside. Also note that, unlike the previous problem, this problem is unrelated to encapsulation: Now, it is not third parties that can modify state in unintended ways, but it is the primary contract itself (which *must* be able to modify the state in the token contract that conceptually belongs to it) whose call has unintended consequences.

4.2 Solution

In order to enable modular verification of inter-contract invariants, we build on our existing approach for proving transitive segment constraints, introduce one additional specification construct, and add proof obligations that prevent both of the potential problems mentioned above. More concretely, transitive segment constraints are now (unlike all other specification constructs we introduced) allowed to express inter-contract invariants, i.e. they may now refer to the state of other contracts that are reachable from the primary contract. All existing proof obligations for transitive segment constraints remain, that is, we check that they are reflexive and transitive, prove that they are established by the primary contract's constructor, and verify that each local segment of the primary contract satisfies them. Additionally, interfaces may declare both function postconditions and transitive segment constraints, which all contracts implementing the interface must adhere to.

We now address Challenge 1 by extending interfaces with specifications that provide the missing guarantees: we allow annotating interfaces with novel *privacy constraints*, which express which part of the contract state conceptually belongs to the caller of a function and, thus, cannot be freely manipulated by other callers. Essentially, a privacy constraint extends the encapsulation guarantees that already exist for the state of the primary contract to (parts of) the state of a secondary contract. Privacy constraints are segment constraints of the form $\forall a.\text{msg.sender} \neq a \Rightarrow P$, where P is reflexive and transitive. The privacy constraints specified in an interface must be satisfied by *all* functions of a contract implementing the interface, even those not mentioned in the interface.

The privacy constraint $\forall a.\text{msg.sender} \neq a \Rightarrow$ `self.balances[a]` $\geq$ **old**(`self.balances[a]`) on the token interface from Fig. 6 expresses that a caller may increase the balance of any contract, but decrease only its own. Since the `steal` function from Figure 7 violates this property, BadToken is now no longer a valid implementation of the Token interface. In other words, the privacy constraint constitutes a promise that the secondary contract does not contain *any* function that will allow third parties to decrease the auction contract's balance. This is exactly the information needed to prove that calls on the token contract by third parties cannot violate the inter-contract invariant stated at the beginning of this section.

In general, assuming that all secondary contracts are annotated with privacy constraints, we prove that those privacy constraints re-encapsulate the state our invariant depends on as follows: We require that each inter-contract invariant is *stable* under state changes allowed by the privacy constraints of all secondary contracts, i.e. that the privacy constraints forbid all changes that could break the invariant. We formally define the notion of assertion stability in Sec. 6 and illustrate it here with our example: Assume that a function of the secondary token contract is called by a party other than the primary contract. For any local segment of the token contract, the token's privacy constraint shown above guarantees that `self.token.balances[self]` $\geq$ **old**(`self.token.balances[self]`). If, in the old state, the inter-contract invariant `self.highestBid` + *sum*(`self.pendingReturns`) $\geq$ `self.token.balances[self]` held, then the privacy constraint (along with the knowledge that the local segment of the secondary contract cannot directly change the state of any other contracts) implies that the inter-contract invariant also holds in the new state.

Challenge 2 is not addressed by the introduction of privacy constraints; since the caller in this scenario is the primary contract itself, privacy constraints offer no guarantees about the potential behaviour of the secondary contract. To address the second challenge, we require that, in every state where the primary contract calls a secondary contract, *any* changes a local segment of the secondary contract can make cannot break the invariant. That is, the conjunction of the privacy constraints and transitive segment constraints of all *other* secondary contracts, *excluding* the called one, conjoined with information we have about the primary contract, must suffice to show that the invariant cannot be broken by the called secondary contract in the state where the call is made. This criterion can again be expressed as a stability constraint (as shown in Bräm et al. [2021]).

For the invariant we attempted to prove, this criterion is not fulfilled, since, as we showed, the secondary contract can break the invariant. We can, however, prove a weaker invariant that still serves our purpose: we exploit that whenever both contracts' state may be out-of-sync, the lock is set and the auction contract cannot be called. So, we require that the desired consistency criterion (that the auction has sufficient funds to pay its obligations) is true whenever the lock field is not set[6], resulting in the invariant $\neg\,\mathsf{self.lock} \Rightarrow \mathsf{self.highestBid} + \mathit{sum}(\mathsf{self.pendingReturns}) \geq \mathsf{self.token.balances[self]}$.

This invariant actually holds, and makes explicit that distributeExcess can rely on its funds being greater or equal the contract's obligations *only* when the lock is not set. As a result, it will now be impossible to prove a postcondition for this function stating that it only increases pending returns, unless the function is fixed by adding **assert not** self.lock at the beginning.

We can prove our adapted invariant as follows: We show that whenever the primary contract calls the secondary contract, the assertion self.lock holds, which implies that our adapted invariant holds as well. This assertion is independent of the token contract and so cannot be broken by changes to its state, and it is preserved by calls to the primary contract (which we can prove using a suitable function constraint $\mathbf{old}(\mathsf{self.lock}) \Rightarrow \mathsf{self.lock}$); that is, it fulfills our stability criterion.

The proof obligations we have outlined ensure that secondary contracts cannot break inter-contract invariants when called by the primary contract (Challenge 2) or anyone else (Challenge 1). Along with the proof obligations that ensure that the primary contract establishes and maintains the invariant, which we described in the previous section, this is sufficient to guarantee that the inter-contract invariant will hold at the end of every local segment of any contract. In summary, the added proof obligations generalise our previously-introduced specification constructs to verify invariants of collaborating contracts. This verification is fully modular, based on the implementation of the primary contract and specified interfaces for all secondary contracts. It is sound even when these contracts interact with unverified code, and in the presence of arbitrary re-entrancy.

5 RESOURCE-BASED SPECIFICATIONS

The vast majority of smart contracts in some way model resources and resource transfers, such as the token and auction contracts we have seen before. Resources have a number of basic properties that are important for the correctness of every contract that works with them: they *cannot be duplicated*, they *have an owner*, and they *cannot be taken away from that owner without their consent*. Explicitly specifying these properties for every smart contract that uses some sort of resource is possible, but laborious and error-prone. Instead, we propose a dedicated specification and verification technique that has basic resource properties built-in and that offers high-level specification constructs to declare resources and to describe resource transactions. The potential of resource-based reasoning for smart contracts has been recognized before; for instance, Move [Blackshear et al. 2019] has native support for resources in the blockchain and language (but does not have built-in guarantees of all the basic properties mentioned above). Compared to specifications that express resource properties via changes of the contract state, our resource specification system has three main advantages (note that Move, due to it having a different resource model, does not have these three advantages built-in, see Sec. 8):

(1) Safety: Basic properties of resources, such as the fact that they cannot be duplicated and cannot be taken away from their current owner without their consent, are baked into the system. Our verification approach ensures that these properties hold by default, without

[6]This pattern is often used in OO-verification when proving invariants between multiple objects [Leino and Müller 2004].

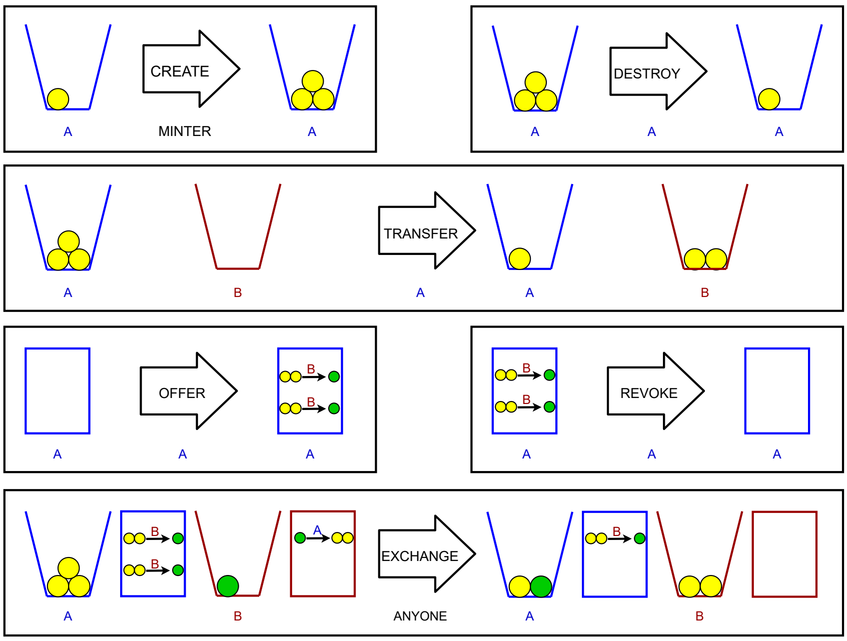

Fig. 8. Operations on resources and offers. Buckets represent resources owned by addresses A and B; the rectangles contain the offers they have made. Big arrows show the name of the resource operation; the names under the arrows show who can perform the operation (create-operations may be performed by anyone who has a special resource representing the right to mint new resources). For example, only A can *transfer* two of its three yellow resources to B, or it can *offer* B to exchange two of its yellow resources against a green one.

developers having to specify them, so that there is no danger that important properties are missing in the specifications, and there is no need to write them down for every contract.

(2) Higher-level reasoning: Developers think about resources as an abstract concept; for instance, they think of a token as a kind of currency, not some contract whose state contains a map. Resource-based specifications let developers describe their contracts' states and interactions on this abstraction level, leading to simpler and more intuitive specifications.

(3) Client documentation: Writing postcondition-based specifications for smart contract functions is often difficult because of potentially re-entrant calls with unbounded effects. Our resource system enables users to prove novel effect-based function specifications that give a caller an upper bound on the negative consequences it may suffer from calling a function (e.g. losing some Ether) and a lower bound on the positive consequences (e.g. receiving some tokens).

In this section, we describe the basic attributes of our resources, the operations that can be performed on them, and how we connect the contract's actual state to the resource state. We show how effect-based function specifications based on resources give callers extra information. Finally, we describe advanced concepts such as *derived* resources, representing titles to other resources.

5.1 Resource Model

In our system, a resource can represent anything that cannot be duplicated, has an *owner*, and has some non-negative value. Resources are owned by addresses on the blockchain. Ownership implies control of the resource, i.e. only the owner of a resource can transfer or destroy it. Receiving additional resources does not require consent. In this regard, our resources are similar to Ethereum's built-in Ether resource (which is treated as a built-in resource in our system).

Fig. 8 shows the operations that can be performed on resources, and who is allowed to perform them. Resources can be *created* by privileged parties who have the right to do so (usually called minters). They can be *transferred* to other addresses or *destroyed* by their owners (and by noone else). In addition, addresses can make *offers* to exchange some resources against others; when an offer from one address exists and a second party makes a matching counter-offer, the *exchange* can be performed at an arbitrary point in time, without further agreement by the involved parties (consuming the offers). For maximum flexibility, an address need not own the resources it offers at the point when it makes the offer, but an exchange requires that both addresses actually hold the offered resources. Addresses can *revoke* previous offers they have made. The resulting set of operations is simple but sufficient to model the behaviour of a wide range of real smart contracts.

5.2 Resource State

Every smart contract may declare one or more resources that they implement (e.g. token contracts would declare a token resource). For each resource, all addresses implicitly have a balance (as for the built-in Ether). Similarly, each address has a set of existing offers on the resources it declares. These balances and offer sets are *ghost state*: state that exists only for verification purposes, but is not present at execution time. Our specifications can refer to this state, e.g. a postcondition can refer to the caller's balance for resource R as $\mathsf{balances}_R[\mathsf{msg.sender}]$. Note that specifications about resource state can be arbitrarily combined with the other specification constructs we have introduced; for example, one could write a segment constraint stating that a contract may perform some operation only if it owns some minimum amount of a resource.

The resource ghost state can be changed only by executing *ghost commands*, written in the verified contract, that each perform one of the resource operations mentioned above. As an example, the ghost command $\mathsf{transfer}_R(f, t, a)$ transfers a amount of resource R from f to t. This ghost command requires that f has sufficient amounts of R, and that f is the contract invoking the ghost command, i.e. that f has called the function that contains it (modulo delegation, which we discuss later). These conditions are checked by the verifier, and they enforce the basic properties of the resource system; the latter check in particular enforces ownership constraints. The resource ghost state has the same encapsulation as ordinary contract state, that is, the ghost commands in a contract can modify only the state of the resources declared in that contract. A more detailed description of the ghost commands from Figure 8 is shown in Bräm et al. [2021].

5.3 Connecting Resource State and Contract State

In order to be useful for verification, the resource ghost state must be connected to the contract's actual state. Our system achieves this by letting developers write invariants (i.e. transitive segment constraints) that relate the resource ghost state (i.e. balances and existing offers) to the contract state. When verifying a contract, we then enforce that these coupling invariants, like all transitive segment constraints, hold at the end of every local segment. For the token contract, the invariant would be $\mathsf{balances}_{token} = \mathsf{self.balances}$, meaning that the balances of the resource named "token" are recorded in the contract's balances field.

This check essentially forces changes on the resource state and on the contract state to happen in lockstep: If a change happens on the resource state with no equivalent change on the contract state (or vice versa), the invariant cannot be verified. As a result, the properties our system guarantees for resources (like ownership) carry over to the actual contract state. For instance, our system prevents the verification of a function steal that allows arbitrary callers to steal some client's funds: the modification to the contract state must be mirrored in a corresponding change of the resource state

```
def transfer(from: address, to: address, amount: uint256):
    { balances_token = self.balances }
    assert self.balances[from] >= amount and msg.sender == from
    self.balances[to] += amount
    self.balances[from] -= amount
    transfer_token(from, to, amount)
    { balances_token = self.balances }
    to.notify(from, self, amount)
```

Fig. 9. Example of contract state and resource state moving in lockstep. In order to re-establish the coupling invariant (blue) at the end of the first local segment of the token contract's transfer function after modifying the contract state, one must execute a transfer command (red) that modifies the resource state accordingly. Verification ensures that all conditions imposed by the resource model are fulfilled when transfer is executed.

(by the coupling invariant). The ghost command that makes this change checks that the transferred funds belong to the function's caller, which would fail here.

As an example, in the transfer function of the token contract, we would have to insert the ghost command transfer $_{token}$(**from**, to, amount) before the call to notify in order to re-establish the invariant, as shown in Figure 9. Verifying the function requires proving that **from** is the msg.sender and that **from** owns at least amount tokens, both of which we can prove because of the assertion at the beginning of the function.

5.4 Client Specifications

The system described so far guarantees that others cannot take away the resources owned by a contract. However, the contract itself may perform operations that lead to a loss of resources, e.g. by transferring or destroying them. Our rules for resource commands ensure that any such operation is initiated by the owner calling a function on the contract that declares the resource (e.g. a function on the token contract that contains a transfer ghost command). Therefore, it is vital that functions provide callers with specifications describing how they affect the caller's resources.

We address this problem by introducing *effects-clauses* on contract functions, which specify ghost commands that will be executed when the function is called (assuming it does not revert). Each function has a multiset of effects, and each effect corresponds directly to one of the ghost commands introduced above, meaning that there are effects for creating, transferring, and exchanging resources, etc. Effects-clauses are unordered and do not give any information regarding *when* during the call the effects occur. As an example, if a function's effects-clause contains transfer $_R$(msg.sender, t, 4) and transfer $_R$(msg.sender, t, 6), this means that after successful execution, it will have transferred 10 R (in two separate steps of 4 and 6) from the caller to address t at some point during the call.

In contrast to traditional effects-systems, our effects-clauses are *not* required to be transitive: the ghost operations performed directly by the called contract's function must be included, but those caused by further external calls made by this contract need not be tracked. This non-standard design is motivated by the presence of unverified code; ultimately, there will be cases in which we could not know the effects of arbitrary external code. The rules for checking effects clauses are simple: (a) the effects-clause of a function *must* contain the effects of all ghost commands directly in its body, and (b) it *may* declare any additional effects resulting from its own calls to other functions.

This might sound problematic: a caller of a function is only able to see the effects clauses (and postcondition) to judge whether they *consent* to these effects happening, but if the effects do not track all transitive calls, one might expect that the caller could be tricked into allowing e.g. more of their resource to be transferred than they realise. Perhaps surprisingly, due to the resource ownership principles built into our methodology, our non-transitive effects actually remain powerful and useful: they ultimately describe worst-case information about what could happen to a caller's resources by calling the function in question *except possibly for additional calls that go via the same original caller.* In other words, the caller is explicitly consenting to at most these effects happening, unless they subsequently consent to additional effects by making a further call.

To see why this is the case, consider a function that may have a negative effect on a calling contract's resources. Since all ghost operations that have a negative effect impose a proof obligation that the resources are owned by msg.sender, the negative effect *must* occur either in the initially-called function, or, after a sequence of additional calls, in some function that was again called *by the original caller.* In the first case, according to check (a), this effect will be included in the initially-called function's effect clause: the caller was aware of the effect and allowed it to happen by making the call. In the second case, for the same reason, this effect must be declared on the subsequently-called function called by the same caller, who consents to the additional effects by making this subsequent call. Note that it *is* possible that a call will cause effects that are positive or neutral for the caller (e.g. an unknown contract giving them tokens) which the called function did not declare; however, since those are not negative for the caller, not knowing about them does not impact the caller's ability to consent to the call.

As a result, our effects-clauses enable each contract to know which negative effects a call may have on its resources, such that it can refrain from making calls with undesired effects. This solution gives strong guarantees in the presence of arbitrary re-entrancy, when it is impossible to give the called function a precise postcondition. Bräm et al. [2021] illustrates the use of effects-clauses on (an extended version of) the token contract from before.

5.5 Derived Resources

As a final ingredient, our system contains one additional concept to model the difference between physically having a resource and conceptually being its rightful owner. As an example, consider the auction contract again: Whenever a bidder sends some wei to it, that wei now physically belongs to the auction contract, which could in principle do with it whatever it wants. However, conceptually, as long as the auction is running, the bidder is still the owner of the wei it sent, and it rightfully expects to either be able to get it back later (if someone else makes a higher bid) or to exchange it for the auctioned good when the auction ends. That is, after a bid and before the end of the auction, the physical owner of that wei (the auction contract) is different from the conceptual owner (the bidder). This is a relatively common notion that occurs whenever some contract (temporarily) manages another contract's resources, and it obviously comes with certain expectations (e.g. the auction contract should not be able to give the wei it has to anyone but its rightful owners).

Our system has support for this scenario in the form of *derived resources*, representing conceptual ownership of a resource physically owned by someone else; essentially, a kind of *title.* In our example, the auction contract could declare a resource wei_in_auction derived from the built-in wei resource, as shown in Figure 10. When a bidder sends wei to the auction contract by calling function bid, it transfers its wei to it, but gets the same amount of wei_in_auction in return, signifying that it is owed that amount of wei from the auction contract. If another higher bid comes in and the bidder gets its wei back by calling withdraw, its titles are destroyed again. In contrast, the winner of the auction exchanges its titles against the auctioned good, so that their bid is now owed to the beneficiary of the auction. At any given point, the amount of titles address c has in the auction contract is $\mathsf{self}.\mathsf{pendingReturns}[c] + (\mathsf{self}.\mathsf{highestBidder} = c ?\, \mathsf{self}.\mathsf{highestBid} : 0)$, meaning that this is also the amount of wei_in_auction contract c owns.

Resource creation and destruction. The existence of an instance of a derived resource is always bound to an instance of the resource it is derived from. That is, if a contract declares resource D derived from another resource R, then an instance of D comes into existence for every instance of

R it receives (via a transfer operation or an exchange), and is automatically allocated to the sender of the R; there is no way to create an instance of D without receiving an instance of R. Similarly, whenever the contract sends some amount of R to someone else, this destroys the same number of D instances that other contract currently owns. This mechanism ensures that the contract always owns enough of the original resource to "pay back" its title loans. The reader may recall that this fact was an invariant of the auction contract that we explicitly mentioned in Sec. 3; now, with derived resources, this invariant is checked automatically and does not have to be specified explicitly.

Resource transfers. In order to ensure that contracts do not give away resources that (according to an existing title) belong to someone else, our system enforces that the contract may now transfer R to another contract *only* if that other contract already owns a sufficient amount of D, i.e. the original contract already owes the second contract at least the amount to be transferred. As an example, when the auction contract sends some amount of wei to a previous bidder of the auction in line 18, this is allowed only if the bidder currently owns an equal amount of wei_in_auction, and if the beneficiary has offered to exchange its wei_in_auction back to ordinary wei. If this is the case, then, the moment the send executes, that amount of the beneficiary's wei_in_auction is automatically destroyed, and the offer to exchange it is consumed.

Apart from the aforementioned restrictions, derived resources behave just like other resources. In particular, they can be traded like other resources (e.g. someone could pay for some good in wei_in_auction, meaning that they give the right to get wei back from the auction contract to someone else). This is relevant for some DeFi contracts that give out tokens that represent ownership of some other goods (i.e. derived resources), but are traded as tokens on their own.

Proof technique. Our proof technique enforces the properties listed above by automatically creating and/or destroying instances of the derived resource whenever a contract calls an external function that declares (in its effects-clause) that it performs a transfer or exchange of the underlying resource to or from the calling contract. Sending or receiving wei is a special case but is treated analogously, i.e. when sending wei, this is handled as if the called function declared that it transfers wei away from the calling contract. To avoid that a contract loses resources that conceptually belong to others without its knowledge (which would mean that it cannot perform the aforementioned checks), our system enforces that the contract declaring D cannot make offers to give away R, since such offers could result in the contract losing R-instances (when the exchange happens) at an arbitrary point in the future.

Bräm et al. [2021] shows the entire auction contract with derived resource specifications.

5.6 Further Extensions

Our system contains a few more features that we are not able to describe fully for space reasons. The most important is the notion of *delegation*: It is sometimes useful or necessary for collaborating contracts to be able to act in each others' names when interacting with other contracts. To enable this, we allow a contract A to decide to *trust* another contract B w.r.t. outside contract C, meaning that when B interacts with C (and only then), it can perform actions that normally only A would be able to perform (e.g. transfer A's resources to someone else). As a result, all restrictions on who may execute certain ghost command that we have discussed so far are implemented modulo trust. Since trusting someone weakens the guarantees one has for one's own resources, users must use this feature with caution; however, as with all other potentially negative effects, functions that establish new trust relations must always state that they do so in their effects-clause.

Our core methodology is easily extended in several further ways; our implementation e.g. has support for resources with identifiers (resources whose instances can be distinguished from one another) useful for modelling non-fungible tokens (NFTs) [Entriken et al. 2018]. Other generalisations are possible, e.g. for some contracts it may be useful to have derived resources that represent

```
resource: good()
resource: wei_in_auction() derived from wei

performs: create[wei_in_auction](msg.value)
performs: offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)
def bid():
    assert block.timestamp < self.auctionEnd and not self.ended
    assert msg.value > self.highestBid and msg.sender != self.beneficiary
    offer[wei_in_auction <-> good](msg.value, 1, to=self.beneficiary, times=1)
    self.pendingReturns[self.highestBidder] += self.highestBid
    self.highestBidder = msg.sender
    self.highestBid = msg.value

performs: destroy[wei_in_auction](self.pendingReturns[msg.sender])
def withdraw():
    pending_amount: wei_value = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, pending_amount)
```

Fig. 10. Example usage of derived resources in a part of the auction contract. Ghost commands are red and specifications like effects-clauses (using the performs keyword) and resource declarations are green. Since the contract declares a resource wei_in_auction derived from wei, sending some wei to it when calling function bid will implicitly create the same amount of wei_in_auction, which then belongs to the bidder. Every bidder offers to exchange their wei_in_auction for the auctioned good if they win the auction. When calling withdraw, previous bidders get back the wei they sent, implicitly destroying their wei_in_auction.

ownership not of a single resource of a different type, but of different amounts of other resources. This feature (like many others) does not have to be built into the system; it can be emulated by using the existing resource model in combination with additional invariants, segment constraints etc. that represent the additional rights and constraints that would result from such resources. As we show in Sec. 7, the set of features we have described gives users a sufficiently expressive model to be able to verify real contracts, while being simple enough for users to reason about.

6 PROOF TECHNIQUE

We summarise here the formalisation of our technique as a separation logic; for space reasons, full details are relegated to Bräm et al. [2021]. We do so for a simple smart contract language reflecting the core of Vyper, with the following commands:

$$c \quad ::= \quad \texttt{skip} \mid x := e \mid \texttt{self}.f := e \mid x := e.\text{fun}(e, \text{value} = e) \mid c;c \mid \textbf{assert}\ e$$

To reflect the design of Vyper only fields of `self` (the current contract) can be assigned; function calls take a second argument representing the amount of wei to send along with a call. We assume a standard expression language with a reserved `result` identifier (to refer to function results in postconditions); field lookups include those on the implicit msg and block arguments. To express two-state assertions such as our segment constraints, our formalisation includes *labels* l denoting earlier points in execution, and expressions $\textbf{old}_l(e)$ denoting the value e had at label l. We use three labels: *pre*, representing the pre-state of the current function, *last*, representing the pre-state of the current local segment, and *call*, representing the pre-state of the last call to another contract.

A state Σ has the form $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, \mathcal{O}, \sigma \rangle$, where $\mathcal{H}$ is the heap (a partial map from contract addresses and fields to their values), and σ is the current variable store. $\mathcal{E}$ is a multiset of *effects* produced so far by the current function and $\mathcal{R}$ is a record containing the state of all resources declared in the current contract. In particular, for every such resource R, $\mathcal{R}.\text{balances}_R$ maps addresses to their balances, and $\mathcal{R}.\text{offered}_{R \leftrightarrow R'}$ tracks the offers to exchange R against another resource R' declared

$$\frac{e_r : T \qquad T.\mathsf{fun}(x) \text{ ensures } Q \text{ performs } S \qquad S' \subseteq S}{\vdash \left\{\begin{array}{c} \mathsf{TSC}[\mathbf{old}_{last}/\mathbf{old}] \\ \wedge \mathsf{SC}[\mathbf{old}_{last}/\mathbf{old}] \\ \wedge e_O \end{array}\right\} \begin{array}{c} x := e_r.\mathsf{fun} \\ (e_a, \mathsf{value} = e_v) \end{array} \left\{\begin{array}{c} \mathsf{perf}(S')[e_r/\,\mathsf{self}\,][x/\,\mathsf{result}\,] \\ [\,\mathsf{self}\,/\mathsf{msg}.\mathit{sender}][e_a/x]* \\ \mathbf{old}_{call}(e_O)\wedge \\ \mathsf{TSC}[\mathbf{old}_{call}/\mathbf{old}]\wedge \\ \mathsf{FC}[\mathbf{old}_{call}/\mathbf{old}]\wedge \\ Q[e_r/\,\mathsf{self}\,][x/\,\mathsf{result}\,] \\ [\,\mathsf{self}\,/\mathsf{msg}.\mathit{sender}][e_a/x]\wedge \\ e_N \Rightarrow \mathbf{old}(e_N) \end{array}\right\}}\ (\mathit{SCall})$$

$$\frac{\begin{array}{c} \mathit{FV}(R) \cap \mathit{mods}(c) = \emptyset \qquad \vdash \{P\}\, c\, \{Q\} \\ R \text{ is stateless if } c \text{ contains a call} \end{array}}{\vdash \{P * R\}\, c\, \{Q * R\}}\ (\mathit{Frame})$$

$$\frac{}{\vdash \left\{\begin{array}{c} e_a \geq 0* \\ (a \neq 0 \Rightarrow \\ \mathsf{trusts}\,(e_f, \mathsf{msg.sender}, \mathsf{true})) \\ *\mathsf{owns}_R(e_f, e_a) \end{array}\right\} \mathsf{transfer}_{\,R}(e_f, e_t, e_a) \left\{\begin{array}{c} \mathsf{owns}_R(e_t, e_a)* \\ (e_a \neq 0 \Rightarrow \\ \mathsf{trusts}\,(e_f, \mathsf{msg.sender}, \mathsf{true})) \\ *\mathsf{perf}(\,\mathsf{transfer}_{\,R}(f, t, a)) \end{array}\right\}}\ (\mathit{Transfer})$$

Fig. 11. Selected Hoare Logic rules; full rules included in Bräm et al. [2021].

in the contract. $\mathcal{R}.\mathsf{trusted}$ is a partial map from pairs of addresses to boolean values that represent whether the first address currently trusts the second; expressions can refer to these maps. Finally, O maps label names to pairs $(\mathcal{H}, \mathcal{R})$ that represent the heap and resource state at label l.

Expression evaluation in a state, denoted by $[\![e]\!]_{(\sigma,\mathcal{H},\mathcal{R},O)}$, is largely standard; most-interestingly, the evaluation of $[\![\mathbf{old}_l(e)]\!]_{(\sigma,\mathcal{H},\mathcal{R},O)}$ is $[\![e]\!]_{(\sigma,\mathcal{H}',\mathcal{R}',O)}$, where $O[l] = (\mathcal{H}', \mathcal{R}')$.

We now define our assertion language as follows:

$$\begin{array}{lcl} P, Q & ::= & \mathsf{emp} \mid e \mid P * P \mid P \wedge P \mid P \mathrel{-\!\!*} P \mid P \vee P \mid \\ & & \mathsf{perf}(E) \mid \mathsf{owns}_R(e, e) \mid \ \mathsf{offers}_{\,R \leftrightarrow R}(e, e, e, e, e) \mid \ \mathsf{trusts}\,(e, e, e) \end{array}$$

Assertion truth in a state is defined by a judgement $\langle \mathcal{H}, \mathcal{R}, \mathcal{E}, O, \sigma \rangle \models P$ whose cases are given in Bräm et al. [2021]. In contrast to traditional separation logics [Reynolds 2002], we do not use the *linear/separating* aspects of the $*$ and $\mathrel{-\!\!*}$ connectives to govern access to the (already-encapsulated) *heap*, but rather for the resource state and effects concepts added by our methodology. The separating conjunction $P * Q$ splits the resource state and the effects into two parts; the first described by P and the second by Q. Descriptions of constituent parts of the resource state come via assertions such as $\mathsf{owns}_R(e_o, e_a)$ that prescribe that $\mathcal{R}$ is empty (no offers, no trust, and all balances are zero) *except* for the balance of e_o, which owns exactly e_a of resource R, and that $\mathcal{E}$ is empty (no effects). The (multiplicative) separating conjunction builds up larger descriptions of these states; e.g. $\mathsf{owns}_R(e_o, e_{a_1}) * \mathsf{owns}_R(e_o, e_{a_2})$ is equivalent to $\mathsf{owns}_R(e_o, e_{a_1} + e_{a_2})$. Similarly, $\mathsf{perf}(E)$ states that *exactly* the effects in multiset E have been performed and no others (and the resource state is empty). The interpretation of other assertions is standard for a classical separation-logic; in particular, an assertion e is true only if there are *no* effects in the current state. As a result, assertions always have to describe the effects-state precisely: If a state containing $\mathcal{E}$ fulfils $P * \mathsf{perf}(E)$, and P does not syntactically contain $\mathsf{perf}()$-assertions, then we must have that $\mathcal{E} = E$. This is important to ensure that functions report all effects they directly cause.

To handle the various kinds of two-state specifications our methodology employs (in each of which $\mathbf{old}(e)$ is used to denote evaluation in the appropriate "old" state), we define a judgement

$\Sigma_1, \Sigma_2 \models P$ in which Σ_1 represents the appropriate state to use as the old one (e.g. for local segment constraints we use the state at the start of the local segment):

Definition 6.1. For two states $\Sigma_1 = \langle \mathcal{H}_1, \mathcal{R}_1, \mathcal{E}_1, O_1, \sigma_1 \rangle$ and $\Sigma_2 = \langle \mathcal{H}_2, \mathcal{R}_2, \mathcal{E}_2, O_2, \sigma_2 \rangle$ we define $\Sigma_1, \Sigma_2 \models P$ if and only if $\langle \mathcal{H}_2, \mathcal{R}_2, \mathcal{E}_2, O_2[last \mapsto (\mathcal{H}_1, \mathcal{R}_1)], \sigma_2 \rangle \models P[\mathbf{old}_{last}(\_)/\mathbf{old}(\_)]$

Using this notion, we can define two-state assertion reflexivity and transitivity as follows:

Definition 6.2. An assertion P is *reflexive* if, for all Σ_0, Σ_1, if $\Sigma_0, \Sigma_1 \models P$, then $\Sigma_1, \Sigma_1 \models P$. An assertion P is *transitive* if, for all $\Sigma_0, \Sigma_1, \Sigma_2$, if $\Sigma_0, \Sigma_1 \models P$ and $\Sigma_1, \Sigma_2 \models P$, then $\Sigma_0, \Sigma_2 \models P$.

We can now also define assertion *stability*, i.e. the fact that an assertion is preserved by another:

Definition 6.3. An assertion P is *stable under* Q, written $\text{stable}(P, Q)$, if, for all $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$, if $\Sigma_0, \Sigma_1 \models P$ and $\Sigma_i, \Sigma_{i+1} \models Q$ for all $i \in \{1, \ldots, n-1\}$, then $\Sigma_0, \Sigma_n \models P$.

Finally, we need to define the notion of a stateless assertion:

Definition 6.4. An assertion P is *stateless* if it does not refer to the current state (including resource state) or an old state *except* for the pre-state (i.e. only old-expressions with label *pre* are allowed).

We define our proof technique via a Hoare Logic formulation, whose details are given in Bräm et al. [2021]. Three important example rules are shown here in Figure 11. Rule (*SCall*) is a simplified version of the most important rule: that for reasoning about calls. Here, ordinary (SC) and transitive segment constraints (TSC) must hold at the end of every local segment; we therefore require them to be true in the precondition of the Hoare triple, using the state labelled "last" as old state. This notion of "last" state is reset in the postcondition to the new current state (we begin a new local segment), as indicated by $e_N \Rightarrow \mathbf{old}(e_N)$ which can be instantiated for any e_N. A similar connection can be made to facts known before the call via e_O.

The frame rule (*Frame*) is non-standard in that it ensures that *no* information about the last state or the current state can be framed around calls; this represents the fact that the entire contract state can change with every call, due to unknown transitive calls. After a call, one may nonetheless assume the transitive segment constraints and function constraints w.r.t. the call's pre-state. To remember information about said pre-state, we use the same trick as before, and allow assuming any expression e_O after a call about its pre-state that was known to be true before the call.

Each ghost command of Sec. 5 gets a corresponding Hoare Logic rule (e.g. (*Transfer*) shown here), which: (a) checks that the participant for whom the command has a negative effect (e.g. giving away resource) trusts the current msg.sender (typically, this is simply who they are), (b) checks that required resources for the command are available, consuming them, (c) adds appropriate new resources in the postcondition assertion, and (d) records the effect that was performed.

The rule for constructors (not shown here) performs the necessary checks of transitivity and reflexivity of transitive segment and function constraints, and ensures that transitive segment constraints fulfil stability criteria described previously.

7 IMPLEMENTATION AND EVALUATION

We have implemented our work in 2Vyper, an automated verification tool for the Vyper language. 2Vyper is open source[7]; it reuses the standard Vyper compiler to type-check input programs. It encodes Vyper programs and specifications into the Viper intermediate verification language [Müller et al. 2016], and uses Viper's infrastructure and ultimately the SMT-solver Z3 [de Moura and Bjørner 2008] to verify the program or otherwise return errors and counterexamples.

[7]https://github.com/viperproject/2vyper

```
contract Client:
  def client():
    self.token.approveAndCall(self.service, amount, data

contract Token:
  def transferFrom(from : address, amount: uint256):
    # transfer 'amount' from 'from' to msg.sender
    # if msg.sender has a sufficient allowance

  #@ performs: revoke[token <-> token](1, 0, to=spender)
  #@ performs: offer[token <-> token](1, 0, to=spender, times=amount)
  def approveAndCall(spender: address, amount: uint256, data: bytes[1024]):
    #@ revoke[token <-> token](1, 0, to=spender)
    self.allowances[msg.sender][spender] = amount
    #@ offer[token <-> token](1, 0, to=spender, times=amount)
    ERC1363Spender(spender).onApprovalReceived(msg.sender, amount, data)

contract Service implements ERC1363Spender:

  def onApprovalReceived(sender: address, amount: uint256, data: bytes[1024]):
    self.token.transferFrom(sender, amount)
    self.performService(sender, amount, data)
```

Fig. 12. Simplified code example showing the functionality of ERC1363 payments. Function approveAndCall also shows the specification syntax used by 2Vyper. The client calls approveAndCall on the token contract and supplies as arguments both the service provider and the input for the requested service. The token contract stores that the service provider may take tokens from the client (in field allowances), and then invokes onApprovalReceived on the service provider, which re-entrantly calls transferFrom to take its tokens and then performs the service. This architecture intentionally uses re-entrancy to allow clients to do in one transaction what would usually require two (one for setting the allowance, one for invoking the service).

While less commonly used than Solidity, Vyper puts a stronger focus on correctness and simplicity, by preventing some errors on the language level (such as over- or underflows, which automatically revert the transaction, unlike in Solidity) and omitting some language features that make code more difficult to reason about (such as inheritance). 2Vyper supports the entire current Vyper language (and several previous versions) and is intended for full-fledged verification of real-world contracts.

2Vyper specifications are written as ♯@ comments in the source code, and use Vyper syntax wherever possible. Fig. 12 shows a simplified excerpt of a function annotated with specifications and containing ghost commands for resource manipulation. In addition to the core correctness properties we have focused on in this paper, 2Vyper also supports reasoning about additional language features (e.g. events) and has additional specification constructs to prove specific kinds of liveness properties (e.g. that auction bidders can eventually get their Ether back; it is not stuck in the auction contract forever) that can be converted to safety properties for verification.

7.1 Evaluation Examples

We have evaluated our approach on a number of real-world smart contracts focusing on existing contracts written in Vyper as well as those involving pertinent features such as inter-contract collaboration or re-entrancy bugs [Arumugam 2019; Blockchains LLC 2016; Ethereum 2021a,d; Minacori 2021; Permenev et al. 2019; Uniswap 2019]. We manually translated some examples without Vyper implementations from Solidity to Vyper.

Table 1 shows the examples; while many consist of a single contract, several either consist of multiple collaborating contracts or of a single contract interacting with external contracts

Application	Contract	LOC	Ann.	IF LOC	IF Ann.	T
auction	auction	63	30	-	-	12.72
auction_token	auction_token††	96	37	88	33	23.67
DAO	DAO*	17	2	-	-	5.13
ERC20	ERC20	98	31	67	25	11.51
ERC721	ERC721	178	32	-	-	15.95
ERC1363	ERC1363	142	31	88	33	22.59
ICO	gv_option_token	98	26	86	36	10.03
	gv_token	121	24	107	49	15.21
	gv_option_program*	86	12	154	67	29.19
	ico_alloc*	159	30	261	116	101.86
Mana	token*	18	3	50	25	2.78
	crowdsale*	42	14	50	25	5.77
	continuous_sale*	36	8	50	25	3.88
VerX_overview	escrow	60	11	65	33	6.36
	crowdsale	41	9	65	33	6.35
safe_remote_purchase	safe_remote_purchase	71	29	-	-	16.84
serenuscoin	serenuscoin	103	4	-	-	6.40
Uniswap V1	Uniswap†	398	115	105	45	112.81

Table 1. Evaluated examples. LOC are total lines of code, *including* specification, excluding comments and whitespace. Ann. are lines of specification. IF LOC and IF Ann. have the same meaning for the interfaces that were required to verify the contract, and T is verification time in seconds. Contracts marked with a star are simplified versions of the original; applications marked with one or two daggers collaborate with an external ERC20 or ERC1363 contract, respectively (accessed through an interface).

via interfaces. We include several examples of complex code used in practice, e.g. ERC tokens, the first version of the Uniswap contract (the largest decentralized exchange and fourth-largest cryptocurrency exchange overall), and an application used to implement the Genesis Vision ICO, which raised 2.8 million USD in 2017. Most contracts were verified in their entirety; for the ICO, we made some small simplifications (in particular, we cut out two option tokens that behaved exactly like a third one and so added nothing of interest); for the Mana and DAO contracts, we focused on specific parts demonstrating inter-contract invariants and a re-entrancy bug, respectively.

7.2 Verified Properties

We now describe the functionality, verified properties, and used specification constructs for some of our examples; if no specific properties are stated, we verified a full functional specification. Bräm et al. [2021] contains descriptions of the remaining examples.

ERC20, auction and auction_token: We have verified an extended version of the auction contract from Fig. 1 and proved all properties mentioned throughout this paper. We have also verified the widely-used standard Vyper ERC20 implementation, which is a more complex version of the token contract in Fig. 2, by declaring a token resource and annotating all functions with the resource operations they perform. We also used segment constraints to specify when the contract triggers *events*, which are a means for the contract to log which transactions have happened in a way that is visible outside the blockchain, and which can easily be specified using segment constraints.

Finally, we have verified a variant of the auction that deals in custom tokens instead of wei against an ERC1363 interface [Minacori 2020] (see below) annotated with resource-based specifications.

DAO: We extracted the buggy part of the DAO contract that led to the loss of ca. 50 million USD [Güçlütürk 2018]. Our implementation declares a derived resource for Ether by default (i.e. it assumes that Ether sent to a contract should still conceptually belong to the sender unless otherwise specified). As a result, when the contract tries to send Ether to an address, an error is reported by default, since our resource model requires the user to justify this action by showing that Ether is only sent to its rightful owner. Since this is not always the case, the contract will be rejected.

ICO: We verified four contracts that implement the Initial Coin Offering (ICO) for Genesis Vision. The ICO progresses in stages, first selling options, then starting the ICO for option holders, and subsequently for the public. Verification required all specification constructs we have presented, e.g. function constraints to describe guarantees made by locks, transitive segment constraints to preserve information across calls (e.g. that the main token, once unfrozen, will never be frozen again), and resource specifications modeling the option token and main token. We used trust to allow that an administrator can freely access other's tokens, which our technique normally rules out. Importantly, we required proving multiple inter-contract invariants to coordinate the four contracts that implement the ICO, e.g. to prove that the main token will be frozen in its first stages.

Some (inter-contract) properties of this example have also been verified in VerX [Permenev et al. 2020]. Notably however, VerX requires the code of all involved contracts at once and does not allow using interface abstractions. In contrast, we use interfaces annotated with specifications to verify each contract modularly. Additionally, while we prove every property proved by VerX, we also proved additional properties (e.g. all standard resource properties such as non-duplicability and ownership, and that the resource operations the contract performs are the expected ones).

Uniswap V1: Uniswap is a popular application that consists of many different exchanges, which together allow clients to exchange different tokens against each other, using Ether as an intermediary. A single exchange is responsible for a single token and, if it wants to buy other tokens, contacts the respective exchange contracts for those other tokens. We declared the desired resource-effects for each function and proved the exchange contract correct w.r.t. them. Again, we did so modularly, using a standard ERC20 interface for its token contract and another interface for other exchanges.

VerX overview: We verified the crowdsale application (consisting of two contracts) from the VerX paper, which again included an inter-contract invariant that we verified *modularly* using interfaces and privacy constraints. Additionally, since one of the involved contracts implements a state machine, we used transitive segment constraints to define valid transitions between states (e.g. once the contract is in the "refund" state, it remains in this state).

ERC1363: ERC1363 is a new token standard [Minacori 2020] that combines into one transaction what ERC20 does in several. Fig. 12 illustrates how this contract intentionally uses re-entrancy in a way that is not ECF and thus cannot be verified using approaches such as VerX.

Conclusion: Our evaluation shows that our specification constructs allow specifying and verifying a wide variety of different properties for real-world contracts. In particular, we can modularly prove inter-contract properties, we can model the resources and resource transactions of different, complex contracts using our resource system (and find typical errors by default), and we can give guarantees for functional correctness and access control even in the presence of unbounded re-entrancy, which allows us to support contracts that employ re-entrancy by design.

7.3 Performance and Annotation Overhead

Table 1 shows the total lines of code of each contract (excluding comments and whitespace, including specification) as well as the lines of annotations we require, and the lines of code and specifications of all interfaces required to verify each contract, as well as the verification time required by 2Vyper. Times were measured by averaging over ten runs, running on a warmed-up JVM.

On our test system (a 12-core Ryzen 3900X with 32GB RAM running Ubuntu 20.04), most contracts can be automatically verified in 5-25 seconds; the two contracts with the longest verification time, both of which are from complex real-world applications, take between 1.5 and 2 minutes.

Considering the strong guarantees afforded by our methodology and tool, we believe even the longest of these times is quite acceptable in practice.

The number of lines required for specifications is less than the number of lines of actual code for every contract. This comparatively modest specification overhead is partly due to our domain-specific resource specifications that allow users to express complex properties in a concise way, and partly due to the design of Vyper, which simplifies verification. Overall, considering the potential financial losses resulting from incorrect smart contracts, writing this amount of specification in exchange for strong functional correctness guarantees is clearly worthwhile.

In conclusion, our technique enables concisely specifying complex correctness properties of (collaborating) contracts, while allowing for modular verification that can be automated efficiently.

8 RELATED WORK

Much recent work has focused on finding problems in smart contracts and proving their absence. Atzei et al. [2017] and Luu et al. [2016] list different kinds of attacks and problems specific to smart contracts. A number of tools have been built to automatically find such problems (e.g. resulting from re-entrancy) using either syntactic patterns [Lai and Luo 2020; Tikhomirov et al. 2018; Tsankov et al. 2018], bounded symbolic execution [Alt and Reitwießner 2018; Luu et al. 2016; Mossberg et al. 2019; Nikolic et al. 2018] or data flow analyses [Feist et al. 2019]. However, most of these tools are unsound by design and can miss errors in real contracts [Feist et al. 2019; Tikhomirov et al. 2018; Tsankov et al. 2018]. Additionally, none of these tools allow proving custom functional properties.

Recent work has studied the difference between harmless re-entrant executions and re-entrancy vulnerabilities [Cao and Wang 2020]. Grossman et al. [2018] have introduced the notion of effectively callback free objects, for which re-entrancy does not introduce any behaviours that are not present in executions without re-entrancy. They provide an algorithm for dynamically checking for ECF-violations and study the decidability of statically proving that a contract is ECF. More recently, Albert et al. [2020] show a static analysis for deciding ECF based on commutativity and projection.

A number of tools aim to achieve verification of custom functional properties for Ethereum contracts, either on the level of the Solidity language [Hajdu and Jovanovic 2019; Kalra et al. 2018] or on the level of EVM bytecode [Hildenbrandt et al. 2018] - to the best of our knowledge, 2Vyper is the first verifier specifically aimed at Vyper. EVM-based verification is not specific to any source language and does not rely on the correctness of the compiler; however, specifications tend to be much more complex on the EVM-level, where high-level abstractions of the source language are lost. Verification tools are either based on SMT solvers [Hajdu and Jovanovic 2019; Permenev et al. 2020], model checking [Mavridou et al. 2019], matching logic [Hildenbrandt et al. 2018], CHC solving [Kalra et al. 2018], or interactive theorem provers [Hirai 2017], which offer different levels of automation and expressiveness. Existing verification tools that offer dedicated, higher level specification languages (e.g. Hajdu and Jovanovic [2019]) typically support single-state contract invariants, but offer no special support for reasoning in the presence of arbitrary re-entrancy beyond that, resulting in imprecision. VerX [Permenev et al. 2020] and VeriSolid [Mavridou et al. 2019] can express temporal properties, which subsume ordinary history constraints; however, VerX explicitly only targets contracts that are ECF, and VeriSolid prevents all re-entrancy by generating code that uses locks throughout. No existing Ethereum verifiers support resource-based specifications.

To our knowledge, the only tools able to prove user-defined inter-contract properties are VerX and VeriSolid. VerX requires the source code of all involved contracts and is therefore not contract-modular, unlike our approach. VeriSolid uses model checking to prove temporal properties on a higher-level model of the contracts and their interactions, and generates correct-by-design Solidity code from this model [Nelaturu et al. 2020]. Unlike our approach, it does not allow reasoning directly about the code of an existing contract (though contracts can be imported into the model).

Researchers have proposed new smart contract languages that aim to simplify verification and/or make it easier to write correct code [Blackshear et al. 2019; Coblenz 2017; Sergey et al. 2019]. In particular, the Move language [Blackshear et al. 2019] offers resources on the programming language level. Unlike our resources, these resources are stateful (in fact, *all* state in Move is stored in resources) and do not have a one-to-one correspondence to physical goods or currency: For example, receiving n coins in Move means adding n to the value stored in one's existing single coin resource, since every address can have at most one resource of every kind. A linear type system ensures that resources are not duplicated in third party code, but the module that defines a resource can modify resource state in arbitrary ways. As a result, incorrect module implementations in Move can violate the properties guaranteed by our resource system (e.g. that resources cannot be taken away from their owners); however, Move's system allows users to manually implement resource models more complex than ours. An SMT-based verifier for custom properties of Move programs exists [Zhong et al. 2020] but currently offers no special support for specifying resource transfers.

To the best of our knowledge, there are three existing approaches for reasoning about (object-oriented) programs in the presence of unverified code. First, Drossopoulou et al. [2020] have introduced *holistic* specifications, which (unlike traditional ones) express *necessary* conditions for an effect to happen, in a setting with arbitrary re-entrancy. They can express e.g. that if a user's token balance decreases, then they either asked to transfer tokens themselves, or another user with a sufficient allowance must have done so. While this kind of property is similar to ones ensured by our resource system, it is not built-in and must be specified manually. Additionally, holistic specifications do not provide support for reasoning about the post-state of calls with arbitrary re-entrancy, and the required (non-standard) reasoning has not been automated, whereas the proof obligations generated by our approach can be checked and automated using standard techniques.

Second, software architectures based on object capabilities [Miller 2006] and object capability patterns [Miller et al. 2000] can be used to encapsulate object state so that properties can be maintained even in an unverified environment. The central idea of object capabilities is to withhold the reference to an encapsulated object from unauthorized third parties, and thereby control who may invoke operations on the object. It is therefore crucial that third parties cannot forge capabilities and thereby obtain unintended access to the encapsulated object. However, since this is not the case in typical smart contract languages (contract addresses are not opaque and can be obtained in various ways, not only by receiving them as an intended capability from another contract), the conditions required for capability-based reasoning are not satisfied in this setting.

Third, Agten et al. [2015] apply separation logic in a context with unverified code by using runtime checks at the boundary between verified and unverified code to ensure that the unverified code has not modified memory it was not permitted to modify. In contrast, our work relies on language encapsulation to ensure this property and therefore does not require runtime checks, which are especially undesirable in a smart contract setting due to the associated gas cost.

9 CONCLUSION

In this paper, we have presented a novel approach for specification and verification of Ethereum smart contracts. Our methodology exploits the features of Ethereum, such as strong encapsulation, to provide guarantees even in the presence of arbitrary re-entrancy, and provides domain-specific specification constructs for resources that make specification both more intuitive and less error-prone. Our evaluation shows that out methology can be implemented efficiently and is capable of expressing and proving complex functional specifications for real-world contracts.

ACKNOWLEDGMENTS

We gratefully acknowledge support from Swiss Stake GmbH.

REFERENCES

Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound Modular Verification of C Code Executing in an Unverified Context. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 581–594. https://doi.org/10.1145/2676726.2676972

Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 209:1–209:30. https://doi.org/10.1145/3428277

Leonardo Alt and Christian Reitwießner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 11247)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 376–388. https://doi.org/10.1007/978-3-030-03427-6_28

Sivakumar Arumugam. 2019. Serenuscoin contract. https://github.com/serenuscoin/contracts Accessed on 2021-04-16.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10204)*, Matteo Maffei and Mark Ryan (Eds.). Springer, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. 2004. Verification of Object-Oriented Programs with Invariants. *J. Object Technol.* 3, 6 (2004), 27–56. https://doi.org/10.5381/jot.2004.3.6.a2

Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. https://developers.libra.org/docs/move-paper

Blockchains LLC. 2016. Decentralized Autonomous Organization (DAO) Framework. https://github.com/blockchainsllc/DAO/blob/6967d70e0e11762c1c34830d7ef2b86e62ff868e/DAO.sol Accessed on 2021-04-16.

Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich Specifications for Ethereum Smart Contract Verification. *CoRR* abs/2104.10274 (2021). arXiv:2104.10274 https://arxiv.org/abs/2104.10274

Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. *Rich Specifications for Ethereum Smart Contract Verification (Artifact).* https://doi.org/10.5281/zenodo.5415274

Qinxiang Cao and Zhongye Wang. 2020. Reentrancy? Yes. Reentrancy Bug? No. In *Dependable Software Engineering. Theories, Tools, and Applications - 6th International Symposium, SETTA 2020, Guangzhou, China, November 24-27, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12153)*, Jun Pang and Lijun Zhang (Eds.). Springer, 17–34. https://doi.org/10.1007/978-3-030-62822-2_2

Michael J. Coblenz. 2017. Obsidian: a safer blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 97–99. https://doi.org/10.1109/ICSE-C.2017.150

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. 2008. A Unified Framework for Verification Techniques for Object Invariants. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5142)*, Jan Vitek (Ed.). Springer, 412–437. https://doi.org/10.1007/978-3-540-70592-5_18

Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12076)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 420–440. https://doi.org/10.1007/978-3-030-45234-6_21

William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. 2018. EIP-721: ERC-721 Non-Fungible Token Standard. *Ethereum Improvement Proposals* 721 (2018). https://eips.ethereum.org/EIPS/eip-721

Ethereum. 2021a. Solidity by example. https://github.com/ethereum/solidity Accessed on 2021-04-16.

Ethereum. 2021b. Solidity documentation. https://solidity.readthedocs.io/ Accessed on 2020-01-11.

Ethereum. 2021c. Vyper documentation. https://vyper.readthedocs.io/ Accessed on 2020-01-11.

Ethereum. 2021d. Vyper example contracts. https://github.com/vyperlang/vyper/tree/master/examples Accessed on 2021-04-16.

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28. https://doi.org/10.1145/3158136

Osman Gazi Güçlütürk. 2018. The DAO Hack Explained: Unfortunate Take-off of Smart Contracts. https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562 Accessed on 2021-03-31.

Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. *CoRR* abs/1907.04262 (2019). arXiv:1907.04262 http://arxiv.org/abs/1907.04262

Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 204–217. https://doi.org/10.1109/CSF.2018.00022

Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10323)*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer, 520–535. https://doi.org/10.1007/978-3-319-70278-0_33

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

Ioannis T. Kassios. 2006. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4085)*, Jayadev Misra, Tobias Nipkow, and Emil Sekerinski (Eds.). Springer, 268–283. https://doi.org/10.1007/11813040_19

Enmei Lai and Wenjun Luo. 2020. Static Analysis of Integer Overflow of Smart Contracts in Ethereum. In *ICCSP 2020: 4th International Conference on Cryptography, Security and Privacy, Nanjing, China, January 10-12, 2020*. ACM, 110–115. https://doi.org/10.1145/3377644.3377650

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. 2008. JML reference manual.

K. Rustan M. Leino and Peter Müller. 2004. Object Invariants in Dynamic Contexts. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3086)*, Martin Odersky (Ed.). Springer, 491–516. https://doi.org/10.1007/978-3-540-24851-4_22

Barbara Liskov and Jeannette M. Wing. 1993. Specifications and Their Use in Defining Subtypes. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993, Proceedings*, Timlynn Babitsky and Jim Salmons (Eds.). ACM, 16–28. https://doi.org/10.1145/165854.165863

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 254–269. https://doi.org/10.1145/2976749.2978309

Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11598)*, Ian Goldberg and Tyler Moore (Eds.). Springer, 446–465. https://doi.org/10.1007/978-3-030-32101-7_27

John McCall, Doug Gregor, Konrad Malawski, and Chris Lattner. 2021. SE-0306: Actors. https://github.com/apple/swift-evolution/blob/main/proposals/0306-actors.md Accessed on 2021-08-07.

Mark S. Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University.

Mark S. Miller, Chip Morningstar, and Bill Frantz. 2000. Capability-Based Financial Instruments. In *Financial Cryptography, 4th International Conference, FC 2000 Anguilla, British West Indies, February 20-24, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1962)*, Yair Frankel (Ed.). Springer, 349–378. https://doi.org/10.1007/3-540-45472-1_24

Vittorio Minacori. 2020. EIP-1363: ERC-1363 Payable Token. *Ethereum Improvement Proposals* 1363 (2020). https://eips.ethereum.org/EIPS/eip-1363

Vittorio Minacori. 2021. ERC-1363 Payable Token. https://github.com/vittominacori/erc1363-payable-token Accessed on 2021-04-16.

Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *CoRR* abs/1907.03890 (2019). arXiv:1907.03890 http://arxiv.org/abs/1907.03890

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, Vol. 2262. Springer. https://doi.org/10.1007/3-540-45651-1

Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Keerthi Nelaturu, Anastasia Mavridou, Andreas G. Veneris, and Aron Laszka. 2020. Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020*. IEEE, 1–9. https://doi.org/10.1109/ICBC48266.2020.9169428

Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 653–663. https://doi.org/10.1145/3274694.3274743

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2019. VerX smart contract verification benchmarks. https://github.com/eth-sri/verx-benchmarks Accessed on 2021-04-16.

Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *PACMPL* 3, OOPSLA (2019), 185:1–185:30. https://doi.org/10.1145/3360611

Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*. ACM, 9–16. http://ieeexplore.ieee.org/document/8445052

Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 67–82. https://doi.org/10.1145/3243734.3243780

Uniswap. 2019. Uniswap version 1. https://github.com/Uniswap/uniswap-v1 Accessed on 2021-04-16.

Fabian Vogelsteller and Vitalik Buterin. 2015. EIP-20: ERC-20 Token Standard. *Ethereum Improvement Proposals* 20 (2015). https://eips.ethereum.org/EIPS/eip-20

Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark W. Barrett, and David L. Dill. 2020. The Move Prover. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 137–150. https://doi.org/10.1007/978-3-030-53288-8_7