# Selective Presumed Benevolence in Multi-Party System Verification

Wolfgang Ahrendt[1][0000−0002−5671−2555]
Gordon J. Pace[2][0000−0003−0743−6272]

[1] Chalmers University of Technology
ahrendt@chalmers.se
[2] University of Malta
gordon.pace@um.edu.mt

**Abstract.** The functional correctness of particular components in a multi-party system may be dependent on the behaviour of other components and parties. Assumptions about how the other parties will act would thus have to be reflected in the specifications. In fact, one can find a substantial body of work on assume-guarantee reasoning with respect to the functional aspects of the component under scrutiny and those of other components. In this paper, we turn to look at non-functional assumptions about the behaviour of other parties. In particular, we look at smart contract verification under assumptions about presumed benevolence of particular parties and focussing on reentrancy issues — a class of bugs which, in the past few years, has led to huge financial losses. We make a case for allowing, in the specification, fine-grained assumptions on benevolence of certain parties, and show how these assumptions can be exploited in the verification process.

**Keywords:** Smart contracts · Interactive systems · Non-functional specifications.

## 1 Introduction

Verification of a component in a distributed system which brings together multiple actors comes with the need for assumptions about how the other parties will interact with the component under scrutiny. In a setting where the other components are processes designed to work well with our code (whether its another method we wrote, a library we trust or a service we rely on), we would typically assume their well-behaviour, effectively presuming their benevolence. In an open-world setting, however, when a component interacts with external parties on whom we have no control, we may have to assume these parties to be malicious. Design of such systems and their verification typically takes one of these two extreme views.

Now consider smart contracts — essentially executable code running on a decentralised platform [12,13] regulating the behaviour of multiple parties. The correctness of such code is, indeed, of high importance to a user of such a system. Bugs may affect functionality, interfering with the user (and other parties) and hindering them from interacting through the smart contract in the originally intended manner. Take, for instance, a smart contract allowing parties to purchase, sell and transfer tokens, and a bug which may occasionally create a new token with a new owner but reusing an existing id, effectively stopping the current user from exercising their rights on their token. Other bugs may be better described as vulnerabilities, not deviating from expected behaviour when used normally, but allow a malicious party to capitalise on them to carry out logic not as originally (hopefully) intended by the smart contract developer. Many such types of vulnerabilities are known in the smart contract world: integer over- and underflow, transaction order dependency, reentrancy, gas-limit attacks, unexpected reverts, etc., with some being easier to avoid than others. Although there is no line, fine or otherwise, to draw between these two types of bugs, the main difference between the two would lie in the *intention* of the party triggering the bug.

Take reentrancy, which we discuss in more detail later on in this paper, and which we will use as an illustrative use case for our contribution. Informally, a smart contract is vulnerable to reentrancy bugs if it exhibits wrong behaviour when the smart contract invokes another party, which unexpectedly calls back the original smart contract.[3] This was the cause of various high-profile instances of bugs, the exploitation of which led to huge financial losses [5]. A verification engineer would not be fazed by such bugs, reasoning that having identified a class of such undesireable behaviour, we can write an anti-specification to capture instances of such bugs. But it is not as simple as this. Sometimes (albeit rarely), reentrancy may be a desireable feature, and in others it may not lead to any undesireable behaviour, making such blanket analyses undesireable.

Furthermore, such a vulnerability may not necessarily be a show-stopper for a particular party. For instance, if we know that there is such a vulnerability which only we can exploit, we may still decide to participate as part of the smart contract (which may already be in place, and cannot be updated). After all, we trust ourselves not to exploit the loophole if the otherwise normal behaviour of the smart contract is what we are after. Consider a smart contract which allows a party to send funds to another party with an intermediary performing due diligence checks on the two parties. If such a contract has a reentrancy vulnerability exploitable only by the receiver resulting in the loss of the funds, and someone is willing to send us funds through the smart contract, we can simply accept and use the contract in the expected way, not exploiting the bug.

This notion can be extended to situations in which we may assume certain other parties not to be malicious.[4] Consider a smart contract which handles the sale of a property, involving the seller and their spouse, the real-estate agent, and

[3] The design of the execution platform on blockchains such as Ethereum, even transferring funds from one smart contract to another involves a call to the recipient's code, providing potential for reentrant calls.

[4] We are sorely tempted to simply say that we *trust* those other parties. However, in the blockchain world, the word *trust* has been heavily used and comes with much baggage — trustless computation, trusted oracles, trusted parties, etc. We thus chose

the buyer. The smart contract to be used may have reentrancy issues which only the seller and/or their spouse may exploit. This may certainly be an issue for the buyer and for the real-estate agent, but from the the seller's point-of-view, if they are happy to assume that neither they nor their spouse will exploit the vulnerability they would not object to the use the smart contract and enabling the transaction to go through. Similar situations may arise in which the parties who may exploit such code would have other incentives, economic, social or otherwise, outweighing their gains from exploiting the smart contract, in which case another party may choose to presume that they will not carry out malicious actions. We propose verification techniques verifying a multi-party system against a specification whilst selectively presuming benelevolence of certain parties.

These are the sort of scenarios we will explore in this paper, essentially proposing a langauge semantics which allows for proving the correctness of specifications annotated by the set of parties presumed to be benevolent — essentially having such *presumptions becoming an element of the specification.* This paper is driven by the idea that the distinction of who we will presume will act in good faith should be made explicit in the specification, and capitalised in the verification. In particular, we will illustrate these notions by focussing on verification of smart contracts written in Solidity [6], explicitly making assumptions about party-benevolence with regards to reentrancy attacks.

**Related work.** Different variations of relative correctness have been studied extensively in the literature (even if 'relative correctness' is not regularly used as a term). In particular, assume-guarantee reasoning has been invented [7,10] to allow compositional verification of component based software. The approach typically taken is to specify functional-correctness properties by writing specifications that make explicit both the assumptions that the component can make about its environment, and the guarantees that it provides. In contrast, in our approach we talk about the *functional guarantees* made by a component, but against *non-functional assumptions* about other components. The use of such compositional verification for smart contracts has not, to our knowledge, been explored, despite the fact that smart contracts are typically multi-party systems. If one focuses on vulnerability due to reentrancy attacks, the literature is rife with techniques and tools to identify them e.g. [9,8,11]. However, all these techniques simply identify the possible reentrancy, and are not concerned with which parties may (or may not) initiate the according attack. Ironically, based on our experience in the sector, it is not uncommon that when faced with such a potential vulnerability, a domain expert may dismiss the concern due to the fact that the only party that can trigger the 'attack' is a trusted party, e.g., the sole owner of a wallet contract, or generally some party that has no interest in running the attack.

The paper is organised as follows. In Section 2 we provide the necessary background about smart contract programming and reentrancy required for the rest of the paper. In Section 3 we present previous work on proof rules for

to avoid the use of the term in order to avoid confusion and talk about *benevolence* and *trust that a particular party will not perform a reentrancy attack.*

Solidity taking a best or worst case approach to reentrancy (i.e. either trusting everyone not to perform a reentrancy attack, or not trusting anyone). Based on this, in Section 4 we present a new proof rules parametrised by presumed party-benevolence, and we finally conclude in Section 6.

## 2 Smart Contracts, Solidity and Reentrancy

Agreements between autonomous agents identify obligations and rights of those agents. Automation of the actions regulated by the agreement partially addresses this concern, in that if actions (and the lack thereof) are enforced by the execution engine, the original agreement is guaranteed to be adhered to. We say that this only *partially* addresses the problem, since a party which can exercise control over the execution platform can still interfere with adherence to the contract. With the development of blockchain and other decentralised ledger technologies, the technology was adapted to provide trustless decentralised execution platforms on which smart contracts can be hosted. On public blockchains such as Ethereum [13], miners validate and carry out transactions initiated by users of the blockchain, recording them in a *block* in return for a mining reward. It is up to the other nodes on the network to ensure that only valid blocks are accepted. The initialisation and execution of smart contracts take the form of special transactions, also executed by a miner writing that transaction in a block, and checked to be a faithful execution by the others. However, unlike a transaction consisting solely of a transfer of funds, executing arbitrary code is not a constant cost operation. The solution typically used is to providing the miner with a reward proportionate to the cost of the computation. This is done by associating each virtual machine instruction with a number of *gas* units, and having the initiator of the call pay a fee to cover the gas required to execute the code. The miner takes funds to cover the total number of gas units consumed by the execution of the call. If the fee does not cover the whole call, it is reverted, but the miner still gets to keep the fee.

Solidity [6] is an imperative programming language (with a sprinkle of object-oriented features) designed and developed to write smart contracts, originally for the Ethereum [13] blockchain. A Solidity smart contract consists of a set of functions each of which can be invoked by parties through a blockchain transaction.

```
contract MultiWallet {
   private mapping (address => uint) balances;

   function depositTo(address receiver) payable public {
      balances[receiver] += msg.value;
   }

   function withdraw(uint amount) public {
      require (balances[msg.sender] >= amount);
      msg.sender.transfer(amount);
      balances[msg.sender] -= amount;
   }
}
```

**Listing 1.** A multi-user wallet written in Solidity.

Listing 1 shows a simple multi-user wallet programmed as a smart contract, which allows multiple users to keep cryptocurrency (in the smart contract) and pay other parties (directly from the smart contract), with a separate account held per user. The state of the smart contract is a mapping from addresses of owners to balances — crypto value is represented as an unsigned integer in Solidity. Two functions are provided, `depositTo()` to allow depositing funds into an individual's account, and `withdraw()` to send funds from a user's account in that smart contract to their address on the blockchain.

At a glance, one may see no features distinguishing the language from the myriad of other imperative or object-oriented languages computer scientists are familiar with. There are, however, certain features which particularly characterise smart contract programming languages:

– Firstly, smart contracts typically handle cryptocurrency (or some form of digital assets), and programming languages for smart contracts provide native functionality to receive and send funds. It is worth noting that smart contracts can own cryptocurrency themselves, which is how the `MultiWallet` contract works. Any funds received are kept in the smart contract and any payment is done by sending funds from the smart contract to the recipient. Solidity provides a `payable` keyword to annotate functions, such as `depositTo()` which, when called, may also receive funds (unless a function is annotated to be `payable`, calls made to the function carrying funds would fail at runtime). For instance, `depositTo()` which is used to put crypto in a user's account, will update the mapping for the identified user by the amount of funds sent upon calling the function. Note that in Solidity, information about the call is stored in the `msg` structure which includes, amongst others, the field `value` which provides the amount of funds sent with the call. To send funds *out* of the smart contract, Solidity uses a method `transfer()` called on the recipient's address.
– The other feature of smart contracts is that they (usually) act as multi-party computation, and thus have an inherent notion of parties. On Ethereum, these parties are identified by their address which corresponds to where a smart contract is located, or to a personal address at which a user can accumulate and control funds. In the `MultiWallet` smart contract, the addresses are used as the key of the mapping to keep track how much of the funds kept in the smart contract are owned by each user.[5] They are also used in the `require` statement (in the `withdraw()` function), which checks that the caller of the function (identified by another field in the `msg` structure — `msg.sender`) has enough funds accumulated in the smart contract to send on.[6]

[5] Note that this allows both humans and other smart contracts to use this smart contract, the former through transactions and the latter through direct calls to the `MultiWallet` smart contract.

[6] If the condition appearing in the `require` statement is not satisfied, the whole computation is aborted and reverted, i.e., any changes to the state are reversed and only a record that a reverted call was made is kept on the blockchain.

We have kept this example simple in order to illustrate the language, but one could have additional logic, for instance releasing the funds only once a trusted third party has checked the identity of the sender or the receiver of the funds.

One distinguishing feature of how smart contracts on Ethereum, and indeed most other Distributed Ledger Technologies (DLTs), work is that a transaction is performed in isolation of others. In other words, if a user sends a transaction to `MultiWallet.depositTo()`, and another user sends one to `MultiWallet.withdraw()`, one of the two transactions is executed to completion before the other can be started. This ensures a degree of atomicity of computation in that function executions do not overlap. However, if a smart contract performs a call to another smart contract, that smart contract may call back the original smart contract, violating this illusion of atomicity. The original transaction will still have been executed to completion, but halfway through, the smart contract was reentered, thus executing another call, possibly even to the same function, within that transaction (on a different level of the call-stack). Such external calls thus require additional care in order to ensure smart contract correctness.

The snag is that when the `transfer` command is used to send funds to another smart contract, this performs an external call to the `receive()` function on the receiver's smart contract,[7] thus ceding control to the other party, who may call back the original smart contract. The simple and innocent-looking code shown in the `MultiWallet` smart contract, in fact, does not take this into account.

Consider a situation in which `depositTo()` is called with Alice's address, sending 1 Ether[8] in the process, and once again with Bob's address, also with 1 Ether. This results in the smart contract storing a total of 2 Ether with the table of balances showing that Alice and Bob own 1 Ether each. If `MultiWallet` is correct, neither Alice nor Bob should be able to use more than 1 Ether from the smart contract. Now consider a call to `withdraw(1 Ether)` coming from Bob's address. Since Bob's balance is 1 Ether, the `require` statement succeeds, allowing the transfer to be called. Recall that if Bob's address points at a smart contract, this is a call to the `transfer` function in that smart contract, which may, apart from receiving the funds, maliciously call `MutliWallet.withdraw(1 Ether)` again. Note that Bob's balance is still recorded as 1 Ether, thus allowing the sending of 1 more Ether to Bob. If Bob's transfer function now just accepts the funds, the calls terminate successfully with Bob having taken 2 Ether out of the `MultiWallet` smart contract, even though he owned only 1 Ether. This class of bugs, called *reentrancy bugs*, have been the cause of many high profile major cryptocurrency losses due to incorrect code, perhaps the most notorous being The DAO[9] hack [5].

[7] In older versions of Solidity, it was the (nameless) fallback function which was executed on the receiver side of a transfer.

[8] Ether is the unit of cryptocurrency on the Ethereum blockchain.

[9] *The DAO* was a smart contract implementing a venture capital fund in a decentralised manner (hence the name DAO referring to Decentralised Autonomous Organisation).

In this case (and as typically recommeded in most Solidity developers' guidelines), one solution is to update the balance *before* the transfer is performed, thus avoiding reentrant calls abusing of the balance not yet being reduced. The following example shows that this rule-of-thumb is not a foolproof one.

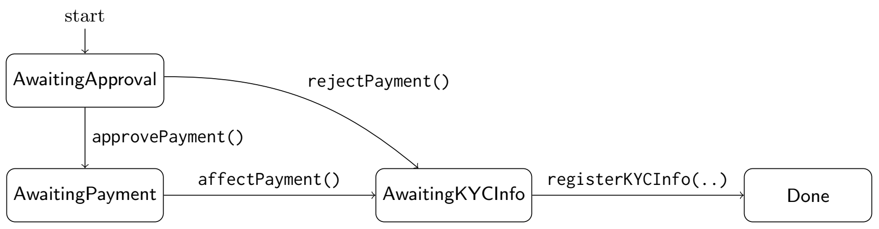

**Fig. 1.** Process flow of a one-shot KYC-approved transfer.

*Example 1.* Consider a smart contract to handle a one-shot payment which depends on an intermediary oracle to perform checks on the proposed receiver of the funds. In practice, such know-your-customer (KYC) checks are required for regulatory purposes, and this contract enables compliance to such requirements by having the sender identify a KYC oracle whose responsibility is to approve or reject the transfer based on background checks they may perform. In return, the oracle smart contract gets paid a commission. The abstract process flow of the smart contract is shown in Figure 1.[10] The smart contract allows the processing of a single transaction, the details of which are assumed to be given in the constructor. Once initialised, the KYC oracle can approve and affect the payment (along the transitions labelled with calls to `approvePayment()` and `affectPayment()` in sequence), or reject it (along the transition labelled with `rejectPayment()`). Whether the oracle approves or rejects the transaction, they are then expected to register the KYC information (along the transition labelled `registerKYCInfo(..)`. These calls (and transitions) implicitly carry information and transfer of funds not shown in the diagram: (i) upon contract setup, the *sender* sends funds into the smart contract by calling the (payable) constructor, providing the *receiver*, the *intermediary* (the KYC checker), the amount to be paid to the receiver, and the amount due as commission to the intermediary; (ii) the intermediary can then approve (using `approvePayment()`) or reject (using `rejectPayment()`) the payment, getting half of their commission; (iii) in case of a rejection the sender gets their funds back immediately, whilst in the case of an approval, the receiver can then withdraw their funds using `affectPayment()`; (iv) at the end of the process, the intermediary can record (a hash of) the KYC information to prove that they really did perform background

[10] The full code of the smart contracts used in this paper can be found in the public repository `github.com/gordonpace/reentrancy-bugs`.

checks using `registerKYCInfo(..)` receiving the rest of their commission.[11] Listing 2 shows select parts of the code. Note that in order to keep track of the state of the smart contract, a variable `mode` is used, ranging over an enumerated type. Also note the constructor which is called when the smart contract is set up.

The contract uses the generally accepted rule-of-thumb of performing transfer of funds at the end of functions, after all variables have been updated. So can anything go wrong due to reentrancy? Upon rejecting a payment using `rejectPayment()`, the intermediary and the sender receive some funds, but upon inspection, one can see that the intermediary can use the fact that the mode of the contract has already been updated, and when their `transfer` function is called, they immediately call `registerKYCInfo(..)` to steal *all* the remaining funds including those of the sender. Therefore, in this example, doing all assignments before the transfers does not prevent the possibility of a harmful reentrance attack!

```
contract KYCTransfer {
  constructor (
    address payable _receiver, uint _payment_amount,
    address payable _intermediary, uint _commission_amount
  ) payable { ... }

  function approvePayment() public byIntermediary { ... }

  function rejectPayment() public byIntermediary {
    // Make sure that we are in the right mode and move
    // on to the mode awaiting KYC information to be recorded
    require (mode == Mode.AwaitingApproval);
    mode = Mode.AwaitingKYCInfo;

    // The intermediary gets 50% of their commission (rounded
    // down if necessary)
    intermediary.transfer(commission_amount / 2);

    // The payer gets his money back
    payer.transfer(payment_amount);
  }

  function affectPayment() public byReceiver { ... }

  function registerKYCInfo(bytes32 _KYC_information) public byIntermediary {
    // Check that the contract is in the right mode and move
    // on to the final mode where no other action is possible
    require (mode == Mode.AwaitingKYCInfo);
    mode = Mode.Done;

    ...
    // Reset payment and commission amounts for safety
    payment_amount = 0;
    commission_amount = 0;

    // Send the remaining commission funds to the intermediary
    // (avoiding rounding calculations)
    intermediary.transfer(address(this).balance);
  }
}
```

**Listing 2.** Code of a smart contract implementing a one-shot KYC-approved transfer.

[11] In practice, we would also need functionality to deal with the case of the intermediary or the receiver not performing their part, introducing timeouts after which the other party can force the logic of the smart contract forward.

An interesting observation can be made here, one that is at the heart of this paper: from the point-of-view of the sender the contract is unsafe, but from the intermediary's point-of-view it can be considered safe. Obviously, it depends on what we mean by safe. In this case, we can specify an invariant which states what funds are expected to be sent and received to and from which party. For instance, one part of the invariant could be that the intermediary will never receive more funds than the amount specified in the constructor. Taking a worst-case scenario (i.e. allowing all parties to perform reentrant calls) this invariant does not hold, although it holds if we take a best-case scenario view (i.e. we assume that no party will perform reentrant calls). However, we can also look at what happens if we only assume that certain parties will not perform a reentrant call. If we only trust the intermediary not to perform such call (either because we are the intermediary or perhaps because the intermediary has too much at stake to perform such attacks), the invariant can be proved to hold no matter what other parties do. Trusting the sender and receiver not to perform reentrant calls, however, does not suffice to enable the invariant to be proved.

**So why does it really matter?** The root problem indicated by the above examples is that both the developer and the reader of a contract's implementation often focus on the main purpose of `transfer`, which is the passing of crypto funds to some receiver. They may not consider the possibility of the receiver calling back the sending contract during the execution of `transfer`.[12]

The fact that this feature led to a number of high-profile losses on Ethereum, resulted in a change in the way the blockchain execution engine handles smart contract fund transfers — there is a cap on the amount of gas which calls to `transfer` can use. The notion of *gas* as a resource to be paid for to execute code is the most common way through which public blockchains motivate miners (the nodes in the decentralised network which process transactions and record them on the blockchain) to execute and record execution of smart contract code. The execution of every operation of the (compiled) smart contract code uses gas, to be paid by the caller. Since many versions of the Ethereum protocol (the latest is called Arrow Glacier at the time of writing), the gas fee of `transfer` is lower than the fee of a call. Thereby, the gas passed to the receiver of `transfer` is not sufficient for the receiver to call any other contract. Any such call would result in the whole call reverting. In particular, this means that it cannot call back.

There are, however, various arguments why one still should consider potential callbacks during `transfer`. One is that the current protection against callbacks depends entirely on the fees for `transfer` and for calls. For instance, a well established Ethereum community blog [1], emphasises that gas fees can and will change, and that *"smart contracts should be robust to this fact"*. But even with stable gas fees, the concern would not go away. The `transfer` operation is actually only one way to pass crypto funds. Another is to explicitly use the `call` operation. For instance, instead of `c.transfer(1 Ether)`, one can write `c.call{value: 1 Ether}("")`. This passes the same `1 Ether` to `c`, but with the

[12] Some are even unaware that any code at all is executed on the receiver side during execution of `transfer`, in case the receiver is a contract address.

difference that all remaining gas is passed on, which makes callbacks possible. Although less safe, this is widely used, even in the standard Solidity library collection *openzeppelin* (e.g., in `utils/Address.sol`). Moreover, a blog [1] from Consensys Diligence, an important player in the Ethereum community, heavily advocates the use `call` instead of `transfer`.[13]

Even if one trusts the call-back protection induced by the gas fee of `transfer`, one just has to replace every `transfer` in this paper by a `call`, and the callback protection is gone. The main reason why we use `transfer` in this paper rather than `call`, is one of simplicity of presentation.

Although we have focussed on the reentrancy in smart contracts, we note that the problem is just one instance of a much more general issue. In all computational contexts where we have a multi-party code base, and where interacting parties may have potentially conflicting interests, there is the question of what can and cannot be assumed of other parties' behavior, and what the consequences of such assumptions are on the safety of one's own code.

## 3 Semantics of Solidity and Handling of Callbacks

There are many reasons why smart contracts are a killer application for formal verification [4]: potentially conflicting interests between participating parties, the immutable nature of the code, the handling of digital assets of value, etc. In [3], a program logic calculus was presented for a core fragment of Solidity. The calculus is implemented in the (currently prototypical) program verification system SolidiKeY, a version of the KeY system [2] dedicated to Solidity verification.

In this section, we will briefly present the calculus rules for `transfer` (with minor adaptions for presentation) in two different versions: (i) a version allowing for the possibility of callbacks during the call to `transfer`; and (ii) a version which assumes no such callbacks. In the previous work [3], the two rules served as a means of performing a best case or worst case scenario verification. Such an upfront decision can be motivated by technical arguments (like the aforementioned gas fees) or as a choice between trusting *everyone*, or *no one* to not call back during a transfer. We will use this as a baseline on which we will later on build rules for handling presumption of *selective* benevolence in later sections.

The KeY approach is based on *dynamic logic* (DL), an extension of first-order logic (FOL) by a *multi-modality* that is parameterised by programs. DL formulae are defined inductively, with FOL connectives as the base forms. The additional form in DL is the *box* modality: $[\pi]$`post`, which takes a program $\pi$ (in our context Solidity code), and another DL formula `post`. The formula means: *if* $\pi$ terminates successfully (i.e., it does not revert), then the property `post` holds in the reached final state. Due to this inductive definition, DL formulas are closed under the usual propositional and first-order connectives. A frequently

[13] The rationale behind their advice is that by not having the safety net of a gas limit, developers are bound to design smart contracts more carefully. The authors disagree. This is akin to saying that one should not wear a helmet on a motorbike, because that would mean that you will drive more carefully.

encountered DL formula pattern using these operators is $\mathsf{pre} \to [\pi]\mathsf{post}$, which corresponds to the Hoare triple: $\{\mathsf{pre}\}\pi\{\mathsf{post}\}$. DL formulas are evaluated in *states*, which are essentially a mapping from program variables to values. A DL formula is valid if it eveluates to true in all states.

In KeY, DL is further extended with *updates* for symbolic execution. An update $v_1 := t_1 \parallel \ldots \parallel v_n := t_n$ consists of program variables $v_1, \ldots, v_n$ and terms $t_1, \ldots, t_n$ (where $n$ can be 1), and represents a state change in the form of explicit substitutions. If $u$ is an update and $\phi$ is a DL formula, then $\{u\}\phi$ is also a DL formula, with $\{v_1 := t_1 \parallel \ldots \parallel v_n := t_n\}\phi$ evaluated in state $s$ being equivalent to $\phi$ evaluated in a state $s'$, where $s'$ coincides with $s$ except that each $v_i$ is mapped to the value of $t_i$ in $s$. In a proof construction, a DL formula of the form $\{u\}[\pi]\mathsf{post}$ represents the status of symbolic execution where the a prefix of a program (path) has already been turned into update $u$, and the remaining program $\pi$ is still to be executed (symbolically). At intermediate proof steps, we can have DL formulae with nested updates like in $\{u_1\}\{u_2\}[\pi]\mathsf{post}$, which will be merged to a single update by further proof steps. See [3,2] for details.

To reason about the validity of formulae, we use a *sequent calculus*, which we present here in a simplified fashion. A *sequent* $\phi_1, \ldots, \phi_n \Longrightarrow \psi$ has a set of formulas $\phi_1, \ldots, \phi_n$ on the left and a single formula $\psi$ on the right[14] and whose meaning is equivalent to $(\bigwedge_{i=1..n} \phi_i) \to \psi$. Calculus rules are denoted as rule schemata of the form:

$$\mathsf{name}\quad \frac{\overbrace{\Gamma_1 \Longrightarrow \phi_1 \quad \ldots \quad \Gamma_n \Longrightarrow \phi_n}^{premises}}{\underbrace{\Gamma \Longrightarrow \phi}_{conclusion}}$$

where $\Gamma, \Gamma_i, \phi, \phi_i$ denote sets of formulas or just formulas, respectively.

Since one important feature of smart contracts is that they can receive, store and send crypto assets, in specifications and proofs it is frequently required to express properties about the asset flow. In order to do so, Solidity DL as presented in [3] has the notion of financial *net* — the difference between incoming and outgoing payments which a contract has with another address. Solidity DL uses the built-in function $\mathsf{net} : \text{address} \to \mathbb{Z}$ to keep track of the asset flow between this contract and other addresses, with:

$$\mathsf{net}(a) = \text{money received from } a - \text{money sent to } a$$

The verification methodology is based on contract invariants relating the external flow of money with the persistent internal data of the contract (which we call *storage*). These contract invariants are expected to hold whenever control is outside the contract. For instance, in the setting of an auction contract where every user can make bids, increase bids, and withdraw bids, the contract invariant

[14] The implemented calculus allows for more than one formula on the right-hand side of a sequent, but here we limit ourselves to one formula for simplicity of presentation.

$I$ may look something like:

$$I \quad \equiv \quad \forall a \in \text{address}.\ \texttt{bids}[a] = \mathsf{net}(a)$$

This invariant states that `bids` is a mapping with the current effective bid of an address $a$, equivalent to the financial `net` of the contract with address $a$.

With this notation, we are now in a position to present the proof rules for the symbolic execution of the `transfer` function in the code of the sending contract.

**Rule for `transfer` with callback.** We start by looking at the version which allows for callback, making no assumption about the receiving contract. Since the transfer may result in callbacks, we must ensure that the smart contract invariant holds before the transfer is made in order to ensure that any callback will find that the invariant holds, and thus also allows us to assume that the invariant will still hold when control is returned after the transfer:

$$\textsf{transfer (with callback)}\quad \frac{\begin{array}{c}\mathit{\Gamma}, \mathsf{c} \neq \texttt{this} \Longrightarrow \{u\}\{\texttt{net(c)} := \texttt{net(c)} - \mathsf{v}\}I \\ \mathit{\Gamma}, \mathsf{c} \neq \texttt{this} \Longrightarrow \{u\}\{\texttt{storage} := st \parallel \texttt{net} := n\}(I \to [\omega]\phi)\end{array}}{\mathit{\Gamma}, \mathsf{c} \neq \texttt{this} \Longrightarrow \{u\}[\texttt{c.transfer(v);}\,\omega]\phi}$$

The rule handles a transfer to the destination address `c` other than this same contract `this`, and allows us to reach a conclusion that $\phi$ will hold after executing `c.transfer(v);`$\omega$, if we start with $u$ as the result of symbolic execution so far. The rule splits the proof into two branches establishing that: (i) after sending the requested amount to `c` (i.e., after substracting `v` from the financial `net` this contract has with `c`), invariant $I$ of contract `this` holds, thus ensuring that the contract is in a consistent state when control is handed over to `c`; and (ii) if the invariant holds, but making no assumption about the `storage` and `net` variables[15] (since callbacks may have changed these values in the meantime), then executing $\omega$ will ensure that $\phi$ will hold upon termination.

**Rule for `transfer` without callback.** For the second version of the rule, which assumes that the receiver of the transfer will not perform a callback, we know that this contract will not be accessed and therefore we need not prove the invariant but simply that $\phi$ will finally hold. This makes the rule substantially simpler:

$$\textsf{transfer (without callback)}\quad \frac{\mathit{\Gamma}, \mathsf{c} \neq \texttt{this} \Longrightarrow \{u\}\{\texttt{net(c)} := \texttt{net(c)} - \mathsf{v}\}[\omega]\phi}{\mathit{\Gamma}, \mathsf{c} \neq \texttt{this} \Longrightarrow \{u\}[\texttt{c.transfer(v);}\ \omega]\phi}$$

In this case, we simply need to prove that, after updating the net of the receiving address, executing $\omega$ to completion will ensure that $\phi$ holds at the end.

[15] The variable `storage` and the mapping `net` are assigned newly generated constants $st$ and $n$, about which no knowledge can be inferred. After that update, the invariant $I$ is assumed, meaning the invariant is all we know about `storage` and `net` when continuing symbolic execution of $\omega$.

In SolidiKeY, the user can configure which rule to use for a proof as a whole, thus deciding *upfront* to either allow potential callbacks from *all* contracts (with callback), or from *none* (without callback). Examples which were verified using either of the rule settings are discussed in [3].

## 4 Presumption of Benevolence

The original motivation for having two different rules was to allow the user of SolidiKeY control whether to trust gas fee-based callback prevention. However, as we have already discussed, the choice can also be made with regards to assumptions about the benevolence of certain parties, where benevolence of a receiver `c` of a transfer `c.transfer(v)` is taken to mean that $c$ will not call back into the sender (directly or indirectly) during execution of the transfer.

Presumption of benevolence can be either *static*, where a fixed set of addresses are presumed to be consistently benevolent, or *dynamic*, where the addresses to be trusted is determined at runtime and may change throughout the execution. In all cases, the specification has to clearly state the assumption under which a property is expected to hold, hence giving rise to different guarantees under different assumptions.

### 4.1 Presuming Benevolence of Oneself

When a smart contract provides a service which a user may want to access, and with no knowledge about other parties involved in the contract, one may try to prove correctness of a specification using a worst case scenario, in which we allow any transfer to perform callbacks. However, although a particular user may not be able to make any assumption about the behaviour of others, they can make them of themselves (or of a smart contract they control). If it is in the user's interest that the smart contract works as per the given specification, they can attempt to prove correctness by assuming no callbacks from transfers to their address — effectively presuming benevolence of themselves.

The following proof rule allows for the differential treatment between a trusted address $s$ (oneself) and that of others:

$$\mathsf{transferBenevolentOneself}\quad \frac{\begin{array}{l}\Gamma, \mathsf{c} \neq \mathtt{this}, \mathsf{c} = s \Longrightarrow \{u\}\{\mathtt{net}(\mathsf{c}) := \mathtt{net}(\mathsf{c}) - \mathsf{v}\}[\omega]\phi \\ \Gamma, \mathsf{c} \neq \mathtt{this}, \mathsf{c} \neq s \Longrightarrow \{u\}\{\mathtt{net}(\mathsf{c}) := \mathtt{net}(\mathsf{c}) - \mathsf{v}\}I \\ \Gamma, \mathsf{c} \neq \mathtt{this}, \mathsf{c} \neq s \Longrightarrow \{u\}\{\mathtt{storage} := st \parallel \mathtt{net} := n\}(I \rightarrow [\omega]\phi)\end{array}}{\Gamma, \mathsf{c} \neq \mathtt{this} \Longrightarrow \{u\}[\mathtt{c.transfer(v);}\,\omega]\phi}$$

The rule effectively combines the two rules given in Section 3, branching over whether the receiver's address is the trusted caller i.e. $\mathsf{c} = s$ — if it is, the premise from the no-callback rule applies[16] (appearing as the first antecdent of

[16] By 'premise applies', we mean that the branches produced by the other premisses close trivially, as their left sides are `false`.

the rule above), and if not, the two premisses of the possible-callback rule apply (appearing as the second and third antecedents in the rule above).

Assuming benevolence of oneself, whoever the caller of a function, we only rely on the fact that if a transfer is made to the trusted address $s$, there will be no callback into the contract, and symbolic execution proceeds accordingly, with no need to show the invariant at that moment.

## 4.2 Presuming Benevolence of a Static Group

As a generalisation of this rule, it may be reasonable to presume benevolence of several parties. For instance, when buying a house together with your siblings, you may choose to trust the behaviour of yourself and your siblings, but not that of the ones selling the house or that of the estate agent. This is an example of a statically determined trusted group and can be encoded as the following rule:

$$\mathsf{transferBenevolent}(S)\quad \frac{\begin{array}{c}\mathit{\Gamma}, \mathsf{c} \neq \mathtt{this}, \mathsf{c} \in S \Longrightarrow \{u\}\{\mathtt{net}(\mathsf{c}) := \mathtt{net}(\mathsf{c}) - \mathsf{v}\}[\omega]\phi \\ \mathit{\Gamma}, \mathsf{c} \neq \mathtt{this}, \mathsf{c} \notin S \Longrightarrow \{u\}\{\mathtt{net}(\mathsf{c}) := \mathtt{net}(\mathsf{c}) - \mathsf{v}\}I \\ \mathit{\Gamma}, \mathsf{c} \neq \mathtt{this}, \mathsf{c} \notin S \Longrightarrow \{u\}\{\mathtt{storage} := st \parallel \mathtt{net} := n\}(I \rightarrow [\omega]\phi)\end{array}}{\mathit{\Gamma}, \mathsf{c} \neq \mathtt{this} \Longrightarrow \{u\}[\mathsf{c}.\mathtt{transfer}(\mathsf{v});\omega]\phi}$$

The rule is almost identical to the previous one, but trusts a receiver if they appear in a (constant) set $S$ i.e. $\mathsf{c} \in S$.

## 4.3 Presuming Benevolence of a Dynamic Group

The two rules we have just seen require the parties to be trusted to be decided statically a priori, and remain unchanged throughout the execution of the smart contract. In practice, this may be too much of a constraint. For instance, one may want to trust a party which is only known at runtime or which may change e.g. in an auction smart contract, one may trust that the current top bidder has no interest in breaking the smart contract's invariant, but the top bidder is not known before the smart contract starts executing, nor does it remain constant throughout the execution.

If we use a predicate `ben` to characterise which addresses are trusted at any point during execution i.e. an address `c` is trusted if `ben(c)`, we can encode such dynamic presumed benevolence using the following rule:

$$\mathsf{transferBenevolencePredicate}\quad \frac{\begin{array}{c}\mathit{\Gamma}, \mathsf{c} \neq \mathtt{this} \\ \Longrightarrow \\ \{u\}\left(\begin{array}{c}\mathtt{ben}(\mathsf{c}) \rightarrow \{\mathtt{net}(\mathsf{c}) := \mathtt{net}(\mathsf{c}) - \mathsf{v}\}[\omega]\phi \\ \wedge\ \neg\mathtt{ben}(\mathsf{c}) \rightarrow \{\mathtt{net}(\mathsf{c}) := \mathtt{net}(\mathsf{c}) - \mathsf{v}\}I \\ \wedge\ \neg\mathtt{ben}(\mathsf{c}) \rightarrow \{\mathtt{storage} := st \parallel \mathtt{net} := n\}(I \rightarrow [\omega]\phi)\end{array}\right)\end{array}}{\mathit{\Gamma}, \mathsf{c} \neq \mathtt{this} \Longrightarrow \{u\}[\mathsf{c}.\mathtt{transfer}(\mathsf{v});\omega]\phi}$$

Compared to the earlier rules, this one makes a case distinction over the predicate `ben`, which may depend on the state. Thereby, the set of trusted parties is not statically fixed. The rule thus has only one premise, meaning it does not cause an immediate branching of the proof since the code which was (symbolically) executed before the transfer, and which is captured in the update $u$, can potentially influence the validity of $\texttt{ben}(\texttt{c})$. The rule will still, indirectly, cause a branching of the proof, which is however delayed to after applying $u$ to the conjunction in further proof steps.

The predicate `ben` will be defined in the specification of the contract. For instance, the contract at hand may have a mapping `trustworthy` from addresses to `bool` to indicate which parties may be trusted. The specification of the contract could define `ben` as follows:

$$\texttt{ben}(a) \;\equiv\; (\texttt{trustworthy}[a] = \texttt{true})$$

As another example, one may trust all addresses which have invested more than 2 Ether into the contract, in which case `ben` would be defined as follows:

$$\texttt{ben}(a) \;\equiv\; (\texttt{net}(a) \texttt{ >= 2 ether})$$

As a final example, we may return to trusting the current top bidder in an auction smart contract by defining `ben` to be:

$$\texttt{ben}(a) \;\equiv\; (a = \texttt{top\_bidder})$$

## 5 Collaborative Malicious Behaviour

One important property of these semantics is that they are monotonic under presumed benevolence. In other words, an invariant which can be proved when trusting a set of parties would still hold if we trust even more parties. For instance, if we write $C \vdash_B I$ to indicate that smart contract $C$ satisfies invariant $I$ if parties $B$ are presumed to be benevolent, we should be able to show that:

> *Program invariants are monotonic with respect to the set of presumed benevolent parties. Given sets of parties $B$ and $B'$, smart contract $C$ and invariant $I$, if $B \subseteq B'$ and $C \vdash_B I$, then it follows that $C \vdash_{B'} I$.*

A natural question is whether we can also use proofs with different sets of parties to require trust in less parties. For instance, if we know that $C \vdash_{B_1} I$ and that $C \vdash_{B_2} I$, can we argue that we do not need to presume benevolence of $B_1 \setminus B_2$ since they are not presumed to be benevolent in the second assumption, nor that of $B_2 \setminus B_1$ since they are not presumed to be benevolent in the first assumption. In other words, does it follow that $C \vdash_{B_1 \cap B_2} I$? We show that this is not the case by counter-example, showing a system and an invariant which holds if we presume benevolence either of party $p_1$ or of party $p_2$. However, we show that that the invariant will not hold if we cannot presume benevolence of anyone i.e. $\{p_1\} \cap \{p_2\}$.

*Example 2.* Consider a two-party token system, in which token ownership by one party indicates certain rights of that party, impinging on the other party e.g. use of a single parking space exclusive to the two parties. An implementation of this as a smart contract is shown in Listing 3, and cycles across three phases (i) the first in which anyone may load the smart contract with funds using `loadAndLaunchTokens()` (e.g. the funds could be jointly contributed to by the two parties, or by their employer); (ii) the second phase in which the parties may acquire the tokens using `acquireToken1()` and `acquireToken2()`, and implemented in such a manner that whenever one party takes a token, the other is compensated in funds; (iii) the third phase in which owners may use the tokens.

```
contract TwoUtilityToken {
    /*@ contract_invariant
        !(tokens_owned1 == 2 && tokens_owned2 == 2) &&
        tokens_owned1 <= 2 && tokens_owned2 <= 2
    */
    ...

    function loadAndLaunchTokens() public payable { ... }

    function acquireToken1() public {
        // Only user1 may invoke this
        require (msg.sender == user1);
        // The contract must be in token acquisition mode
        require (mode == Mode.TokenAcquisition);

        // User 1 acquires one token
        tokens_owned1++;
        // Pay user 2 compensation
        user2.transfer(1 ether);

        // Move on to the token usage mode if enough tokens have been acquired
        if (tokens_owned1 + tokens_owned2 >= 2)
          mode = Mode.TokenUsage;
    }

    function acquireToken2() public { ... }

    function useToken() public { ... }
}
```

**Listing 3.** Code of a two user utility token.

The smart contract invariant will ensure that the supply of tokens is limited — it will never be the case that any one of the users owns more than two tokens, or that both users own two tokens each. The implementation takes a safer route, and switches to token usage mode (phase 3) as soon as the sum of tokens owned reaches two as can be seen in the code shown for function `acquireToken1()`. It is worth noting that the state of the smart contract is encoded in the variable `mode`, which is updated at the *end* of `acquireToken1()` i.e. after transfering funds, because of reentrancy. Since control is temporarily given to user 2 during the transfer of funds, if we would have already updated the mode, that party would have the opportunity to use any token they own *before* user 1 can, thus giving them an undesirable advantage.

The question is whether anything can go wrong. Verifying the invariant using the no-reentrancy semantics allows us to prove that the contract is safe, indicating that if anything can go wrong, it would have do so due to reentrancy. The invariant can also be proved if we assume that the first user will not attempt reentrancy, and similarly for the second party. Good, both primary stakeholders have the power to ensure that the smart contract does not violate the invariant. But does that suffice? Consider the following execution pattern: (i) user 1 acquires one token using `acquireToken1()`; (ii) user 1 calls `acquireToken1()` again to acquire a second token, but when user 2 is given control through the call to `transfer` (on line 19), (iii) user 2 calls `acquireToken2()`[17] to (hurriedly) acquire a token, but as user 1 gains control when their compensation is being transferred, (iv) user 1 will call `acquireToken1()` again. Since the mode is only changed at the *end* of the functions,[18] all tokens will be successfully acquired, resulting in user 1 owning 1 token, and user 2 owning 3 tokens, thus violating the invariant. Furthermore, if gas were not a concern, the two parties can nest this attack to any depth, acquiring even more tokens.

Verifying the smart contract allowing both parties to use reentrancy would, in fact, identify this vulnerability. But is the contract safe to use if violation can only occur through collaboration? It largely depends on other parties involved in the contract. If a third party e.g. the employer of the two users is the one loading the smart contract with funds to pay compensation, then the parties might collaborate together to take more funds than originally envisaged. However, if the parties are the only ones involved (they are also the ones who would load funds into the smart contract), then the contract is safe in that either party can play clean to ensure that the invariant is not violated. This illustrates an interesting game theoretic situation in which we would be able to reason about conditional contract safety with respect to an invariant as long as we have a notion of user benefits and gains.

## 6 Conclusions

We have looked at a particularly thorny issue of reentrancy in smart contracts — a vulnerability that has led to huge financial losses in the past few years. In fact, the problem is a much more general one, that of reasoning about systems which perform synchronous external calls and may thus have to handle callbacks when ceding control to these other parties. Typically, such systems are designed with their specifications holding only when they can trust certain other parties never to perform such callbacks, which raises the issue of how to deal with specifications whose correctness may be conditional on other parties. In a way, it is a problem akin to assume-guarantee, but with the assumptions being non-functional ones i.e. not about what properties hold of the results that the external party will give us, but rather about how that party will compute that result. Combining such functional and non-functional aspects of a specification in order to enable formal proof is not straightforward, as we have seen in this paper.

[17] The code of `acquireToken2()` is identical to `acquireToken1()` but with the parties switched.

[18] As already discussed, moving it forward in the code will open another vulnerability.

One aspect of smart contracts which we have alluded to multiple times in this paper is that of parties' interest. Since smart contracts explicitly encode the notion of parties and of digital asset migration, one can typically encode some form of utilitarian interpretation to a particular smart contract state change being to the benefit of a particular party but possibly not to that of another. Such an interpretation would allow for economics-based reasoning of whether a particular party stands to gain or lose from a callback. This would enable a more refined version of callback-freedom semantic analysis: assuming rational agents, and thus that a party would never engage in callbacks if they stand to lose if they do so, does a smart contract satisfy a particular specification under such an assumption? The notion of a party's interest in a contract is not always straightforward to encode (e.g. is the value of gaining 10 tokens alone any different from that of yourself and another party gaining 10 tokens each?) and we leave this as future work.

## References

1. Consensys Diligence: Stop using Solidity's transfer() Now. `https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/`
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, LNCS, vol. 10001. Springer (2016). `https://doi.org/10.1007/978-3-319-49812-6`
3. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 9–24. Springer (2020). `https://doi.org/10.1007/978-3-030-61467-6_2`
4. Ahrendt, W., Pace, G.J., Schneider, G.: Smart contracts: A killer application for deductive source code verification. In: Müller, P., Schaefer, I. (eds.) Principled Software Development. Springer (2018)
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: POST. LNCS, vol. 10204, pp. 164–186. Springer (2017)
6. Ethereum: Solidity. Online Documentation `http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html` (2016)
7. Jones, C.B.: Development Methods for Computer Programs Including a Notion of Interference. Ph.D. thesis, Oxford University, UK (1981)
8. Li, B., Pan, Z., Hu, T.: Redefender: Detecting reentrancy vulnerabilities in smart contracts automatically. IEEE Transactions on Reliability **71**(2), 984–999 (2022). `https://doi.org/10.1109/TR.2022.3161634`
9. Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B.: Reguard: Finding reentrancy bugs in smart contracts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion). pp. 65–68 (2018)
10. Misra, J., Chandy, K.: Proofs of networks and processes. IEEE Transactions on Software Engineering **7(7)**, 417–426 (1981)
11. Mueller, B.: Smashing Ethereum smart contracts for fun and real profit. In: HITB SECCONF Amsterdam (2018)
12. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. White Paper `https://bitcoin.org/bitcoin.pdf` (2009)
13. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)