Proof-Carrying Smart Contracts

Thomas Dickerson[1], Paul Gazzillo[2], Maurice Herlihy[1], Vikram Saraph[1], and
Eric Koskinen[2]

[1] Brown University
[2] Stevens Institute of Technology

Abstract. We propose a way to reconcile the apparent contradiction between the immutability of idealized smart contracts and the real-world need to update contracts to fix bugs and oversights. Our proposal is to raise the contract's level of abstraction to guarantee a specification φ instead of a particular implementation of that specification. A combination of proof-carrying code and proof-aware consensus allows contract implementations to be updated as needed, but so as to guarantee that φ cannot be violated by any future upgrade.

We propose proof-carrying smart contracts (PCSCs), aiming to put formal correctness proofs of smart contracts *on the chain*. Proofs of correctness for a contract can be checked by validators, who can enforce the restriction that no update can violate φ. We discuss some architectural and formal challenges, and include an example of how our approach could address the well-known vulnerabilities in the ERC20 token standard.

1   Introduction

Motivation.   The promise of smart contracts seems impossible to fulfill. In theory, a smart contract is a transparent agreement, freely agreed upon by informed parties. Irrevocable and immutable, it enforces itself without need for help from humans and their civic institutions. In reality, an unhappy history of exploits, theft, and fraud has established that people are bad at writing correct contracts, and no better at detecting flaws in the contracts they agree to [17, 6, 18, 5].

The Solidity language and EVM bytecode permit contracts to call code at a dynamic address. While intended to support legitimate functions such as sending a payment to a user or contract, it allows the contract to implement the *pointer to implementation* (or *PIMPL*) idiom. This idiom has one benefit: it provides a path through which a buggy contract implementation might be patched. Even though the *code* is immutable, the *state* managed by the code, including the implementation pointer, is not. The danger, of course, is that a dishonest party could use such dynamic control flow to make substantial changes to the contract's terms after it has been agreed upon.

We believe this dilemma can be avoided, or at least mitigated, by including formal correctness proofs in the blockchain itself. Suppose we identify a property φ critical to the contract's integrity. If a flawed implementation C that formally satisfies φ can only be replaced by an improved implementation C' such that

$C' \vDash \varphi$, then all parties to the contract can be confident that φ will continue to hold even as bugs (not covered by φ) are detected and patched.

The idea of mixing code and proofs goes back to Necula's proof-carrying code (PCC) [16]. The blockchain context, however, brings new challenges: What format is needed for contracts' specifications, and where are these specifications stored? Which code needs to be verified? Who generates the proofs? How do miners repeatedly validate the proofs? Here we sketch preliminary work on the architectural and formal aspects of these challenges, illustrated by a running example example based on the ERC20 token standard [21].

Overview. We believe that one may be able to adapt Necula's PCC into a variation that we call *proof-carrying smart contracts* (PCSC). The idea is that a contract's API and specification φ are published to the blockchain. Any subsequent updates to the implementation C must be accompanied by a valid proof that the specification is maintained: $C \vDash \varphi$. Smart contracts are well-suited to PCC, because a proof only needs to generated once by the contract owner, before the implementation pointer is updated. Publishing the proof to the blockchain makes it immutable, and the participants in the network only need check the proof's validity, a task that is far less computationally expensive than generating the proof.

Blockchains and blockchain consensus require some changes to standard PCC. In the original formulation, a code *consumer* specifies the specification, and the code *producer* generates a proof that the policy is preserved by the remote code. In the blockchain context, the smart contract owner is the code producer. All validators and clients in the blockchain network are code consumers, since smart contracts are replicated and rerun by all.

We exploit the immutability of blockchain to enable the the producer to provide the specification. Because this specification is published before any updates to the contract implementation, code consumers can inspect it before transacting with the contract. Immutability of blockchain data guarantees the policy can never be weakened by the producer. The producer updates the implementation pointer with a special setter that must be accompanied by a valid proof. Blockchain consensus ensures that updates by the contract writer preserve the specification, as long as a majority of participants validate the proof.

Running Example. The ERC20 specification [21] defines a number of operations intended to provide a standardized API for managing tradeable tokens on the Ethereum platform [7]. Using ERC20, Alice may `approve` up to some number n tokens, the `allowance`, to be transferred to Bob, and Bob may then execute multiple `transfer` calls until all n have been transferred (or Alice reduces Bob's `allowance`). Here is an example, based on a simplified version of the ERC20 operations, how one might use PCSC. In our simplified ERC20 specification, there are only two accounts, `from` and `to`. Token transfers happen only unilaterally, flowing from `from` to `to`.

There are several invariants linking operations i and $i+1$. First, we enforce conservation of tokens, by defining

$$\texttt{total\_supply}_i = \texttt{from\_balance}_i + \texttt{to\_balance}_i \tag{1}$$

and requiring

$$\texttt{total\_supply}_{i+1} = \texttt{total\_supply}_i \tag{2}$$

Second, we enforce the allowance limit:

$$\Delta_i \leq \texttt{allowance}_i \tag{3}$$

$$\texttt{from\_balance}_{i+1} = \texttt{from\_balance}_i + \Delta_i \tag{4}$$

$$\texttt{allowance}_{i+1} = \texttt{allowance}_i - \Delta_i \tag{5}$$

The naïve implementation in Fig 1 appears to respect both invariants. Unfortunately, these conditions, while necessary, are not sufficient to guarantee the expected behavior. Alice, having initially `approve`d an allowance of 100 tokens, may later wish to decrease Bob's allowance by calling `approve(50)`. But if Bob has already executed `transfer(n)` (for $n > 50$), Alice's call has the unexpected effect of *increasing* Bob's total withdrawal.

This is essentially a data race: even though the EVM execution is single-threaded, transactions are submitted in parallel, and miners may reorder and interleave those transactions arbitrarily.

Fig 2 is a version of `approve` that is safe from the data race. It forces the sender to first set the allowance to 0 before updating[3], effectively clearing the allowance before setting a new one.

Contributions This paper makes make the following contributions:

- We propose proof-carrying smart contracts (PCSC), a way to allow contracts to be upgraded while ensuring that critical properties such as φ are preserved.
- We describe an architecture for PCSC, along with a discussion of needed changes to the blockchain protocol and virtual machine (Section 2).
- A treatment of PCSC with specifications and proofs (Section 3).

This paper describes the ideas and insights motivating this work, which is still in progress.

2 Realizing Proof-Carrying Smart Contracts

Adopting the language of PCC, the code *producer* is the contract writer, while the *consumers* are all other participants in the blockchain network, miners, validators, and the clients who issue transactions. The producer's role is to create

---

[3] This unfortunately restricts the possible valid semantics of the ERC20 implementation, later in the paper we will propose yet a 3rd implementation that is thread safe without being subject to this restriction.

```
 1 bool transfer(uint256 value) {
 2   if (value <= allowed) {
 3     from_balance -= value;
 4     to_balance += value;
 5     allowed -= value;
 6     return true;
 7   } else { return false; }
 8 }
 9 uint256 allowance() {
10   return allowed;
11 }
12 // vulnerable to a data race
13 bool approve(uint256 value) {
14   allowed = value;
15   return true;
16 }
```

Fig. 1. First version.

```
 1 bool approve(uint256 value) {
 2   // sender sets allowed to 0
 3   // first to avoid data race
 4   if (allowed == 0 && value > 0
 5      ||allowed > 0 && value==0) {
 6     allowed = value;
 7     return true;
 8   } else { return false; }
 9 }
```

Fig. 2. Altered version of `approve`.

```
 1 bool transfer(uint256 value) {
 2   if (value <= allowed) {
 3     from_balance -= value;
 4     to_balance += value;
 5     allowed -= value;
 6     allowed_known = false;
 7     return true;
 8   } else { return false; }
 9 }
10 uint256 allowance() {
11   if(caller == sender) {
12     allowed_known = true;
13   }
14   return allowed;
15 }
16 bool approve(uint256 value) {
17   // sender must have observed
         allowance
18   if(allowed_known){
19     allowed = value;
20     return true;
21   } else { return false; }
22 }
```

Fig. 3. A safe version of `approve`, emulating LL/SC for `allowed`.

A simplified example of an implementation of the ERC20 token standard where tokens are sent with a two-step `approve`/`transfer` process.

the contract specification, consisting of an API and persistent contract state. Unlike the original formulation of PCC, the specification is provided by the producer, rather than the code consumer, along with the contract code. The specification is provided as invariants on contract state as well as pre- and post-conditions on the API interface methods. The *parent* part of a PCSC includes the smart contract's internal state, API methods, and formal specification. A (successfully deployed) *child* part of the PCSC includes the source code for all API methods, and a proof that the implementation satisfies the specification (found in the corresponding parent).

We exploit the immutability of the blockchain to put executive control of specification in the hands of consumers. In order to run smart contract trans-

actions, the contract is first appended to the chain. By requiring producers to include the specification with the contract, it becomes part of the immutable history of the chain, and can't be modified, even if, for example, the contract uses PIMPL and the implementation pointer changes to a new child contract. Even though the consumers do not directly create the specification, a consumer can inspect the specification and choose not to issue transactions on the contract (though they must still process any transactions issued by other consumers, as long as those transactions obey the specification).

We use the consensus mechanism of the blockchain to ensure that all updates to the implementation of the smart contract preserve the original specification. The producer specifies the fields of any child contract addresses. Any updates to these fields must be accompanied by a proof that the new child contract satisfies the original specification[4]. Participants trust that miners and validators check the validity of proofs, just as they trust them not to zero out someone's token balance or put the contract in some other incorrect state. As with Bitcoin, Ethereum, and other cryptocurrencies, as long as the majority of miners are well-behaved, the contract updates will obey the specification.

Being a manifestation of PCC, the producer generates this proof off-the-chain before making the update. The special setter function, allowing the child address to be updated, uses a new bytecode operation, SAFEUPDATE, to check the provided proof preserves the specification. This ensures that miners and validators can guarantee the new child contract is safe. The code for the child contract must be available before the call to update the child address, so the parent must publish the initial contract before issuing the update or prove an existing contract satisfied the specification [5]. This is safe because contract addresses are not reassigned or unassigned due to the immutability of the blockchain.

As with other violations (out-of-gas, exceeded stack depth), an update to the child address without a proof or with an invalid one is rejected with an exception. The validation of the proof is far less computationally expensive than generating the proof, enabling higher throughput for miners. But due to the added computational expense of validation, this new update operation will require more gas than a typical store operation, perhaps proportional to the size of the proof.

Fig 4 illustrates the operation of proof-carrying smart contracts (PCSC). This diagram assumes the producer has already published the parent contract (containing the API and specification) as well as the candidate child contract's code, because these operations require no verification[6]. The producer first generates a proof that the proposed child contract satisfied the invariants of the specification (Step 1). This proof is packaged in an update transaction. The producer issues

---

[4] For the purposes of this paper, we assume that the compiler is able to translate proofs & invariants for the source language (e.g. Solidity) into proofs & invariants for the host language (e.g. EVM bytecode).

[5] A dummy contract which always terminates with an exception should vacuously satisfy the specification, since, for example, under the Ethereum model of execution, a contract could run out of gas and terminate at any point anyway

[6] Generally-speaking, a parent contract can include arbitrary computation as long as it is accompanied by its own proof.
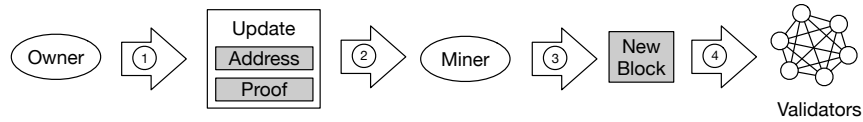
Fig. 4. The proof-carrying smart contract update operation.

the transaction to the blockchain network as usual for mining (Step 2). The miner validates the proof against the code of the proposed child contract and produces a new block containing the safe update (Step 3). If the proof is invalid, the block will still record the attempted update as a transaction, but the safe update will fail with an exception (and consume gas to disincentivize spurious update requests). Finally, the rest of the blockchain network participants rerun and validate the `SAFEUPDATE` (Step 4), as they would any other transaction.

2.1 Proof-Carrying Smart Contracts in Detail

Here we describe the details that a realization of the PCSC architecture entails as well as discussion of generalizations to the architecture.

For many use cases, the parent contract will be nothing more than a thin wrapper for calls to the child contract, and perform no computation other than delegating its method calls. This means that proofs about the behavior of the parent should typically be compact, and in this way the parent contract resembles a formal specification for an API more than it does a fully-fledged smart contract in its own right. Similarly, in the PIMPL pattern, the child contract will typically have no persistent state of its own, operating instead on the state of the parent. If some persistent state is present, e.g., caching of expensive math operations, in the child, it must be guaranteed to not affect the global state invariants of the parent. This organization provides a clean separation of specification and implementation. In the Ethereum virtual machine, the parent contract would use a `DELEGATECALL` to the child contract, to ensure the contract operates on the parent contract state.

In addition to declaring the API specification and persistent state, the parent contract must declare the fields holding the addresses to the child contracts. At the source-code level, this is achieved with a specially declared setter that modifies the addresses. There are several options for ensuring the smart contract cannot subvert proof validation by modifying a child contract address without using the setter. For instance, runtime instrumentation of store operations can ensure nothing touches a child address field. Additionally, the specification itself can describe invariants about updates to the child address field. This latter option is more amenable to source code analysis, rather than byte code analysis, since we can prohibit arbitrary address computation.

We assume that the contract is not valid until provided an initial child contract. Exception handling can be encoded in the specification to ensure correct behavior if a client calls the parent contract before the initial update. As previ-

ously noted, the specification must always be robust to exceptions on platforms where computations (of potentially unpredictable length) must be paid for in advance (e.g. Ethereum). For our PIMPL-based examples, the parent has only one child contract, but it is straightforward to extend this to multiple contracts. The producer identifies the fields of each child contract and must provide proofs when updating each. Again, in our examples, the parent makes no other calls to contracts in our current formulation of simplified ERC20, but there is nothing fundamental preventing multiple child contracts, so long as each child address must be modified only with the safe update command that ensures a proof is provided with the update. Similarly, the child contract may also call yet more contracts, so long as proofs can be generated for their behavior.

As for altering the specification, a more flexible architecture can permit updates to the specification by the contract writer. This would be possible as long as the new specification implies the previous one, i.e., the specification can only become more strict.

Lastly, it is not necessary that the smart contract specification be decided by an individual. A common type of proposal and voting contract can be used to distribute decision-making. Participants can propose and vote on the specification, which is automatically installed by the voting contract. Furthermore, in cases where a specification is for standard behavior that might be incorporated into many contracts, we might imagine this proposal voting system be used to produce standardized APIs. In this way, all blockchain participants are both producers and consumers. Modifications to the specification, under the previously stated implication rule, could be decided in a similar way.

There are two sorts of upgrades that our proof-carrying code scheme permits. *Minor upgrades* can install a new child contract as long as the parent contract's safety policy is preserved. This is enforced by the the proof verification performed by SAFEUPDATE. This permits safe upgrades without involving the slower consensus process required to decide on the safety policy. The proof verification makes this possible without having to trust the developer to maintain the safety policy. This enables upgrades due to minor bugs not covered by the safety policy or performance upgrades for instance. *Major upgrades* can alter the safety policy itself and require consensus among contract participants, e.g., via a proposal and voting system. Like the minor upgrade, the child contract implementation is replaced, but the new safety policy itself is also provided. The SAFEUPDATE verifies the new child contract against the new policy. This requires the new child contract author to generate a proof against the new safety policy.

The tradeoffs here are that maintainers can easily perform minor upgrades whose safety is ensured by the original safety policy of the contract, lowering obstacles to development. But the discovery of limitations of the safety policy itself or a desire for organizational changes may warrant an upgrade to the policy itself.

Let us assume the token contract Fig 1 has been initially installed with only the invariants specified in Eqns 1–5, i.e., the safety policy only guarantees the results of balance transfers, but does not account for the approve/transfer data

race A minor upgrade, such as eliminating a superfluous call to a safe math library, can be performed, since it provably does not violate the safety policy. At a later point, the organization discovers the data race and agrees to update the safety policy to ensure future versions of the contract avoid it. This major update is accompanied with a revised implementation of the contract that satisfies the new invariants. Future contracts can then perform minor maintenance or performance upgrades, like removing safe math calls, that continue to satisfy the safety policy.

3 A Proposal for Specifications & Proofs

In recent years there has been substantial progress on formal verification of smart contracts at both the high-level Solidity language [15, 19] as well as low-level EVM bytecode [12, 11]. We aim to exploit this progress to enrich the blockchain so that (i) smart contract APIs come with formal specifications of how they should operate and (ii) proposed smart contract implementations can include proofs that they satisfy those specifications. In this section, we discuss sketch formal aspects, building on specification formats for objects [3, 2, 13, 8] and Necula's proof-carrying code [16].

As discussed in the prior section, our PCSCs involve two components. The *parent* part of a PCSC includes the smart contract's internal state, API methods, and formal specification. A (successfully deployed) *child* part of the PCSC includes the source code for all API methods, and a proof that the implementation satisfies the specification (found in the corresponding parent). We now provide more detail on each of these, using the running example.

3.1 Parent: APIs & Specifications

State and Methods. The parent in a PCSC includes the *state* in the form of object fields. It could include any of the smart contract language's data-types (integers, strings, Booleans, mappings, arrays, etc.). In the ERC20 running example, the state of the PCSC includes:

$$
\begin{array}{lll}
\texttt{balance}: & Addr \mapsto \mathbb{N} & \text{Relate addresses to balances} \\
\texttt{allowed}: & Addr \mapsto (Addr \mapsto \mathbb{N}) & \text{How much others can transfer} \\
\texttt{child\_ptr}: & Addr & \text{Pointer to implementation}
\end{array}
$$

We have already discussed the purpose of `balance` (a mapping from addresses to tokens, represented as natural numbers) and `allowed` (a mapping from addresses to address-token mappings). The final element of the state above is a critical component of the PCSC parent. It is a pointer to the *child* part of the PCSC which will contain the implementation (discussed below).

The parent in a PCSC also includes the interface, in the form of methods that can be called by participants in the network. The bodies of these methods simply relay the call, following the `child_ptr` to the corresponding method in the child in the PCSC. Here is an example:

```
1 api_transfer(uint256 value, addr from) : bool {
2    return child_ptr.transfer(value, from);
3 }
```

This PIMPL paradigm means that the correctness of the parent contract follows immediately from that of the child (discussed below), modulo initialization concerns. In general, PCSCs needn't necessarily follow the PIMPL paradigm and it is easy to imagine other arrangements. For example, a contract may wish to specialize dispatch, in which case all child contracts would need to be proved correct. For simplicity, we focus on the common PIMPL case in the remainder of this paper.

$\{I\}$
`api_transfer(uint256 value, addr from) : bool`
$\left\{ \begin{array}{l} I \land \Sigma_a \text{ `balance}(a) = \Sigma_a \text{ balance}(a) \\ \quad \land \text{ `allowed(from)}(me) \geq \text{value} \\ \qquad \Rightarrow \text{allowed} = \text{`allowed[from}, me \mapsto \text{`allowed(from)}(me) - \text{value]} \land \text{rv} = \text{true} \\ \quad \land \text{`allowed(from)}(me) \leq \text{value} \Rightarrow \text{allowed} = \text{`allowed} \land \text{rv} = \text{false} \end{array} \right\}$

$\{I\}$
`api_allowance(addr whom) : uint256`
$\{I \land \Sigma_a \text{ `balance}(a) = \Sigma_a \text{ balance}(a) \land \rho_{me}(\text{whom}) = \text{allowed}(me)(\text{whom})\}$

$\{I\}$
`api_approve(uint256 value, addr whom) : bool`
$\left\{ \begin{array}{l} I \land \Sigma_a \text{ `balance}(a) = \Sigma_a \text{ balance}(a) \\ \quad \land (\text{`allowed}(me)(\text{whom}) = \rho_{me}(\text{whom}) \\ \qquad \Rightarrow \text{allowed} = \text{`allowed}[me, \text{whom} \mapsto \text{value}] \land \text{rv} = \text{true}) \\ \quad \land (\text{`allowed}(me)(\text{whom}) \neq \rho_{me}(\text{whom}) \Rightarrow \text{allowed} = \text{`allowed} \land \text{rv} = \text{false}) \end{array} \right\}$

where I is the global invariant, defined to be:

$\forall a.\text{balance}(a) \geq 0 \land \forall a\, b.\text{allowed}(a)(b) \geq 0 \land \forall a.\text{balance}(a) \geq \Sigma_b \text{ allowed}(a)(b)$

Fig. 5. Formal specification φ (in blue) for some of the ERC20 token standard.

Formal Specifications. The parent in the PCSC also contains the specification φ of how the overall PCSC is intended to behave. As we will discuss later, a candidate child implementation C is required to include a proof that $C \vDash \varphi$.

Fig. 5 provides an example specification φ for a portion of the (simplified) ERC20 token standard that we are using as a running example. Each method API includes standard Floyd-Hoare style *pre*-conditions as well as *post*-conditions, depicted in blue. For a given method, say, `api_transfer`, the meaning is that, if we assume that the associated pre-condition holds before `api_-`

`transfer` executes, then a correct implementation will ensure that the corresponding post-condition must hold upon completion. (We assume that every method will terminate, because the smart contract architecture enforces termination through "gas.") Each pre/post-condition includes I which is a global invariant on the state of the PCSC. I is defined at the bottom of Fig. 5. There are three conditions given by I: that all balances are non-negative, all allowances are non-negative, and that, for a given address a the sum of all outstanding allowances is bounded by a's balance (respectively). The latter condition corresponds to Eqns. 3-5 in Section 1.

In the specification for `api_transfer`, me is used to denote the caller's address and notation '`allowed` indicates the value of `allowed` before the method executed. The post-condition for `api_transfer` includes a stipulation that the sum of all participants' `balance`s is unchanged (corresponding to Eqns. 1 and 2 in Sec. 1). We use a as a quantifier variable over each participant's address. The post-condition includes two further cases, depending on whether the transfer request is permitted by `allowed`. If it is permitted, then the value of `allowed` is the same as '`allowed`, except that the appropriate slot is decremented. Otherwise, `allowed` is unchanged. `rv` indicates the return value of the method.

Specification for `approve` *.* The published ERC20 standard has a well-publicized flaw, demonstrated by the naïve implementation in Fig. 1, which is that calls to `approve` a new allowance do not impose any particular semantic requirements on the previous value. Thus an account holder may inspect the blockchain, and see a current allowance value and attempt to reduce it at the same time that another transaction is issued to transfer some of it. Since pending transactions are subject to arbitrary reordering by the miner, the transfer may execute first, and altered allowance may have the net effect of raising the total that can be transferred.

Conceptually, we wish to add another invariant: the allowance may not be altered unless the allowance is known *when the transaction executes* (this may be different than the value it had when the transaction was issued). The implementation shown in Fig. 2 patches this vulnerability by requiring that a new positive allowance can only be set if the allowance is currently being set to 0 (either by `transfer`s or by `approve`s). This blocks the data race, but also forces the account holder to pay for unnecessary transactions when a competing transaction is *not pending*.

Multiprocessor architectures address similar data race problems with atomic instructions such as *compare-and-swap*. To fix the ERC20 API, however, it is more convenient to mimic the functionality of *load-linked* (LL) and *store-conditional* (SC) instructions. `LL` loads a value from memory, and `SC` writes a new value to the same location, if and only if it has not been written since the matching `LL`.

Our specification in Fig. 5 includes the requirement that `allowance` is known at the time of approval, using a ghost variable. The specification for `api_-allowance` uses ghost variable ρ in the post-condition. This variable tracks the fact that the caller (me) has checked how much recipient `whom` is currently per-

mitted to transfer. ρ_{me} can become out-of-date if the recipient makes a call to `transfer`, and this will be the saving grace in the specification of `api_approve`. In the specification for `api_approve`, `allowed` is updated, approving a pending recipient `whom` to receive `value`. The two cases depend on whether ghost variable $\rho_{me}(\texttt{whom})$ is up-to-date, indicating that me is aware of how much has been approved.

In the next subsection, we will discuss how the implementation using strategy employed in Fig. 3 can be proved to satisfy this specification.

3.2 Child: Proposed Implementations & Proofs

Miners propose the child portion of a PCSC: an implementation C, coupled with a proof that the implementation satisfies the specification φ housed in the parent. We now discuss what the child portion of the PCSC entails.

Implementation C. The implementation C of each API method (`transfer`, `allowance`, etc.) is housed in the child PCSC such that, if C can be shown to satisfy φ, then the child will be installed and these implementations will be accessed via `child_ptr.transfer()`, etc.

Proof that $C \vDash \varphi$. How can we be 100% sure that this proposed implementation in Fig. 3 operates correctly? The child portion of a PCSC includes a proof that code C satisfies the corresponding parent's specification φ.

The Floyd-Hoare style pre/post specifications shown above can be verified to hold of an implementation using verification conditions as seen in tools such as Spec# [3], Boogie [2], Dafny [13], Why3 [8], etc. Intuitively, the format of the proofs are, for each line of each method, invariants that must hold at that line (more precisely: the invariant comes just before or just after the line). Finding these invariants is difficult (searching for a proof). Checking these invariants, however, is much faster: a symbolic analysis can traverse the method, starting by ensuring that the first invariant holds from the the pre-condition and effect of the first line of code. When the analysis reaches the end of the method, it checks that the post-condition is entailed by the penultimate invariant and last line of code.

PCSCs allow us to prevent buggy implementations like `approve` in Fig. 1 from being accepted onto the blockchain, but permit correct implementations like Fig. 3. There is no proof that Fig. 1 satisfies the specification in Fig. 5. On the other hand, a proof can easily be given for Fig. 3, which is a simplified case where there are only two participants and variable `allowed_known` is used to ensure the correct behavior of `approve`.

3.3 Verification Tool Development

In our ongoing work, we are developing verification tools for PCSC, building on recent works for verification of solidity [15, 19] and EVM [12, 11].

Ultimately, the proofs published to the blockchain need to be expressed in terms of the bytecode, to avoid dependencies on a specific verified compiler. Fortunately, others have developed formal semantics for EVM bytecode [12, 11]. Down the road, we plan to extend work on certified compilation [14] to translate source-level correctness guarantees to bytecode guarantees. However, there are already verification challenges at the source code level, being tackled by us and others [15, 19].

4   Related Work

Ethereum's [7] *ERC20* token standard [21] is widely used as the basis for many recent *initial coin offerings*. Vladimirov and Khovratovich [20] give a clear description of the ERC20 design flaw discussed here.

The notion that proofs should be included with code first appears in Necula's seminal proof-carrying code paper [16]. As mentioned, we make use of the functionality of the Why3 platform [4]. Hicks and Nettles [10] pioneered the idea of using PCC for dynamic software updates.

There is other work that investigates vulnerabilities in smart contracts. For example, Luu *et al.* [15] develop a software tool called Oyente, which detects security bugs in Ethereum contracts. Atzei *et al.* [1] describe common pitfalls that lead to security vulnerabilities, and demonstrate how they can be exploited. Sergey and Hobor [17] analyze smart contract vulnerabilities by drawing comparisons between contract execution and concurrent shared-memory computing. Grossman *et al.* [9] discuss a dynamic approach.

5   Conclusion and Future Work

This paper describes preliminary work attempting to reconcile the apparent contradiction between the immutability of idealized smart contracts and the real-world need to update contracts to fix bugs and oversights. Our proposed solution is to raise the contract's level of abstraction to guarantee an invariant φ instead of a particular implementation of that invariant. A combination of proof-carrying code and proof-aware consensus allows contract implementations to be updated as needed, but so as to guarantee that φ cannot be violated by any future upgrade.

Much remains to be done on proof-carrying smart contracts. The work reported here is still in an early stage, and we are not yet far enough along to report on progress or difficulties.

Future goals include formally modeling proof-carrying smart contracts and creating an implementation as an extension of the Ethereum blockchain and virtual machine. A formal specification will permit proofs of guarantees that proof-carrying smart contracts provide. Additionally, we intend to investigate how consensus integrates with these proofs and perhaps extend the model consensus to include them. Extending smart contracts with specifications requires defining extensions to the smart contract implementation language and the bytecode to

represent the specifications as well as mappings from source code to bytecode specifications. For generating and validating proofs, we plan to use off-the-shelf tools, such as Why3. Our language extensions and the proof tools need to be integrated with the smart contract toolchain and virtual machine itself.

For implementation, we intend to extend the contract virtual machine with new opcodes to add new contracts with specifications as well as update them given a new proof. To enable this, we will extend the binary format of smart contract to encode specifications and proofs. Using these changes in an existing chain would require a hard fork to extend the binary format and virtual machine. With proof-carrying smart contracts in hand, we will use them to improve the ERC20 token standard, demonstrated with example implementations, and show how contract writers can take advantage of these.

Proof-carrying smart contracts open up new research questions. For instance, how do we integrate proofs into blockchain consensus and how do mining and consensus mechanisms, such as proof-of-work and proof-of-stake, interact with formal proofs? Formal verification enables trust for updates, but consensus mechanisms are still needed to agree on what the right specifications are. For instance, contract participants can vote on changes to the specifications, but allow formal verification to eliminate the need for voting on implementation changes.

The ability of formal verification to support trusted computing has the potential to improve how consensus is achieved, and proof-carrying smart contracts are an important step in integrating proofs with blockchain.

References

1. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts sok. In: Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204. pp. 164–186. Springer-Verlag New York, Inc., New York, NY, USA (2017). https://doi.org/10.1007/978-3-662-54455-6_8, `https://doi.org/10.1007/978-3-662-54455-6_8`
2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. vol. 5, pp. 364–387. Springer (2005)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. pp. 49–69. Springer (2004)
4. Bobot, F., Fillitre, J.C., March, C., Melquiond, G., Paskevich, A.: The why3 platform. `http://why3.lri.fr/manual.pdf`, accessed: 14 January 2018
5. Daian, P., Breidenbach, L.: Parity proposals potential problems. `http://hackingdistributed.com/2017/12/13/ether-resurrection/`, retrieved 14 Jan 2018
6. DAO: The DAO smart contract, retrieved 8 February 2017
7. Ethereum: `https://github.com/ethereum/`, accessed: 14 January 2018
8. Filliâtre, J.C., Paskevich, A.: Why3: where programs meet provers. In: European Symposium on Programming. pp. 125–128. Springer (2013)
9. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications

to smart contracts. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) (2018)

10. Hicks, M., Nettles, S.: Dynamic software updating. ACM Trans. Program. Lang. Syst. **27**(6), 1049–1096 (Nov 2005). https://doi.org/10.1145/1108970.1108971, `http://doi.acm.org/10.1145/1108970.1108971`

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

12. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: International Conference on Financial Cryptography and Data Security. pp. 520–535. Springer (2017)

13. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 348–370. Springer (2010)

14. Leroy, X., et al.: The compcert verified compiler. Documentation and user's manual. INRIA Paris-Rocquencourt (2012)

15. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, `http://doi.acm.org/10.1145/2976749.2978309`

16. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 106–119. POPL '97, ACM, New York, NY, USA (1997). https://doi.org/10.1145/263699.263712, `http://doi.acm.org/10.1145/263699.263712`

17. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. CoRR **abs/1702.05511** (2017), `http://arxiv.org/abs/1702.05511`

18. Sirer, E.G.: Parity's Wallet Bug is not Alone. `https://blogs.apache.org/foundation/entry/apache-struts-statement-on-equifax` (2017), [Online; accessed 05-Nov-2017]

19. various: Formal verification for solidity contracts. `https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts`, accessed: 14 July 2018

20. Vladimirov, M., Khovratovich, D.: Erc20 api: An attack vector on approve/transferfrom methods. `https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbTOmooh4DYKjA_jp-RLM/edit#heading=h.m9fhqynw2xvt`, accessed: 14 January 2018

21. Wiki, E.: Erc20 token standard. `https://theethereum.wiki/w/index.php/ERC20_Token_Standard`, accessed: 14 January 2018