Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond

Shaun Azzopardi[1], Joshua Ellul[1,2], and Gordon J. Pace[1,2]

[1] Department of Computer Science, University of Malta, Malta
[2] Centre for Distributed Ledger Technologies, University of Malta, Malta

Abstract. Smart contracts present new challenges for runtime verification techniques, due to features such as immutability of the code and the notion of gas that must be paid for the execution of code. In this paper we present the runtime verification tool ContractLarva and outline its use in instrumenting monitors in smart contracts written in Solidity, for the Ethereum blockchain-based distributed computing platform. We discuss the challenges faced in doing so, and how some of these can be addressed, using the ERC-20 token standard to illustrate the techniques. We conclude by proposing a list of open challenges in smart contract and blockchain monitoring.

1 Introduction

Although the general principles of runtime monitoring and verification are well established [15,24,5], applying these techniques and building tool support for new architectures frequently brings to the fore challenges in dealing with certain aspects of the architecture. Over the past few years, the domain of blockchain and distributed ledger technologies has increased in importance and pervasion, and with it came an arguably new software paradigm or architecture, that of smart contracts. Borrowing much from a multitude of existing technologies, including distributed computing and transaction-based systems, brings forth a new set of challenges for runtime monitoring and verification. In this paper, we expose our runtime verification tool ContractLarva for monitoring smart contracts, and discuss the open challenges in adapting dynamic verification for this domain.

The key novel idea behind distributed ledger technologies (DLTs), is how to achieve an implementation of a distributed and decentralised ledger, typically guaranteeing properties such as transaction immutability[3] i.e. achieving a form of ledger synchronisation without the need for central points of trust. Blockchain was one of the first algorithmic solutions proposed to achieve these goals, with the underlying ledger being used to enable keeping track of transactions in Bitcoin [26], the first instance of a cryptocurrency. Since then, various extensions and variants were proposed, with one major idea being that of *smart*

[3] In this context, one typically finds the use of the term *immutability* for immutability of transactions or data written in the past, whilst still allowing for appending new entries (in a controlled manner) to the ledger.

contracts, allowing not only the immutable storage of transactions, but also that of logic which may perform such transactions. Smart contracts thus enable the enforcement of complex exchanges (possibly consisting of multiple transactions) between parties in an indisputable and immutable manner.

Smart contracts in themselves are not a new concept. They were originally proposed by Nick Szabo in 1996 [30] as *"contracts [...] robust against naïve vandalism and against sophisticated, incentive compatible breach."* Szabo's view was that while legal contracts typically specify ideal behaviour — the way things should be — e.g. *"The seller is obliged to deliver the ordered items on time,"* nothing stops the parties involved from behaving outside these bounds[4], a smart contract would *enforce* the behaviour, effectively ensuring that it is not violated in the first place. As most eminently highlighted by Lessig, *"code is law"* [23] — what code allows the parties to do or stops them from doing, effectively acts as theoretically inviolable legislation. The smart contract thus would typically chooses a path of action which ensures compliance with the agreement the parties have in mind. However, nothing stops the party entrusted with executing the code from modifying it or its behaviour, hence there remains the requirement of 4a regulatory structure to safeguard that such modifications do not occur — in practice, simply moving the need for a legal contract one step away.

Blockchain, however, provided a means of doing away with the need for such centralised trust in or legal agreement with the party executing the code, and the first realisation of this notion was the Ethereum blockchain [32], which supported smart contracts in the form of executable code running on a decentralised virtual machine, the Ethereum Virtual Machine (EVM), on the blockchain.

For instance, consider the natural language (legal) contract regulating a procurement process between a buyer and a seller, as shown in Figure 1. Clause 8 states that: *"Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover payment of all pending orders."* This may be achieved in different ways. For example, this may be achieved by receiving payment upon the creation of every new order. However, since (according to clause 3) the buyer will already have put in escrow payment for the minimum number of items to be ordered, one may choose to use these funds as long as there are enough to cover the newly placed order, still satisfying clause 3. The legal contract does not enforce either of these behaviours, but rather insists that the overall effect is that of ensuring funds are available in escrow. In contrast, a (deterministic) executable enforcement of the contract would have to choose one of the behaviours to be executed.

The question as to whether specifications should be executable or not has a long history in computer science (see [17] vs. [20]). Executable specifications require a description of *how* to achieve a desired state as opposed to simply describing *what* that state should look like in a declarative specification — and

[4] In practice, what stops these parties from doing so is the threat to be sued for breach of contract, which happens outside the realm of the contract itself.

1. *This contract is between ⟨buyer-name⟩, henceforth referred to as 'the buyer' and ⟨seller-name⟩, henceforth referred to as 'the seller'. The contract will hold until either party requests its termination.*
2. *The buyer is obliged to order at least ⟨minimum-items⟩, but no more than ⟨maximum-items⟩ items for a fixed price ⟨price⟩ before the termination of this contract.*
3. *Notwithstanding clause 1, no request for termination will be accepted before ⟨contract-end-date⟩. Furthermore, the seller may not terminate the contract as long as there are pending orders.*
4. *Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.*
5. *Upon accepting this contract, the seller is obliged to place the amount of ⟨performance-guarantee⟩ in escrow.*
6. *Upon termination of the contract, the seller is guaranteed to have received payment covering the cost of the minimum number of items to be ordered unless less than this amount is delivered, in which case the cost of the undelivered items is not guaranteed.*
7. *The buyer has the right to place an order for an amount of items and a specified time-frame as long as (i) the running number of items ordered does not exceed the maximum stipulated in clause 2; and (ii) the time-frame must be of at least 24 hours, but may not extend beyond the contract end date specified in clause 2.*
8. *Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover payment of all pending orders.*
9. *Upon delivery, the seller receives payment of the order.*
10. *Upon termination of the contract, any undelivered orders are automatically cancelled, and the seller loses the right to receive payment for these orders.*
11. *Upon termination of the contract, if either any orders were undelivered or more than 25% of the orders were delivered late, the buyer has the right to receive the performance guarantee placed in escrow according to clause 5. Otherwise, it is released back to the seller.*

Fig. 1. A legal contract regulating a procurement process.

the 'how' is often more complex than the 'what'[5], and leaves more room for error.

The possibility of error is a major issue. Smart contracts, being executable artifacts, do exactly what they say they do, but that might not be what the contract should have done. As smart contracts grow in size and complexity, this issue becomes more worrying, and there have been well-known instances of such smart contracts that allow for misbehaviour, for instance, on Ethereum [4]. Ideally, the correctness of smart contracts is verified statically at compile time, but using automated static analysis techniques to prove business-logic level properties of smart contracts has had limited success, with most work focussing on classes of non-functional bugs. This leaves great scope for runtime verification to provide guarantees over smart contracts.

In this paper, we present ContractLarva[6], a tool for the runtime verification of smart contracts written in Solidity, a smart contract programming language originally proposed for the use on Ethereum, but now also used on other blockchain systems. We summarise the salient features of Solidity in Section 2, and discuss the design of ContractLarva in Section 3. Given the immutable nature of smart contracts, bugs can be a major issue since simply updating the

[5] NP-complete problems are a classical case of this — although there is no known deterministic polynomial algorithm which can find a solution to instances of any one of these problems, a known solution to an NP-complete problem instance can be verified in polynomial time on a deterministic machine.

[6] Available from `https://github.com/gordonpace/contractLarva`.

code may not be an option. We discuss how this can be addressed using dynamic verification in Section 4. There are many other open challenges in smart contract monitoring, some of which are discussed in Section 5, while in Section 6 we discuss existing work in verification of smart contracts. Finally, we conclude in Section 7.

2 Smart Contracts and Solidity

Blockchains provide a decentralised means of a shared ledger which is tamperproof and verifiable. Smart contracts built on a blockchain (like Ethereum) allow for decentralised execution of code, which could implement agreements between different parties, similarly in a tamperproof and verifiable manner. Solidity is the most popular language used to write Ethereum smart contracts. Solidity gets compiled down to EVM bytecode — a 256-bit stack-based instruction set which the Ethereum virtual machine will execute. At the bytecode execution level, the EVM can be seen as a 'one world computer' — a single shared abstract computer that can execute smart contract code. Once a smart contract is compiled to EVM bytecode, the contract may be uploaded and enacted on the blockchain having it reside at a particular address, thus allowing external entities to trigger its behaviour through function calls — therefore a smart contract's publicly executable functions represents the contract's Application Programming Interface (API). Functions are atomic i.e. they execute from beginning to end without interruption, and the EVM is single threaded which implies that only one function, or rather one instruction from all smart contracts is executed at a time, even though the EVM is distributed amongst all nodes. This implies that a long running function, or more so a function that never terminates, would slow down or stop all other smart contracts on the platform from executing. To prevent this, Ethereum requires an amount of *gas* to be sent by the initiator to be used to pay for the execution of code. If the gas runs out, then execution stops and all state changed since execution initiation is reverted. This mechanism ensures that infinite loops will eventually stop since the finite amount of gas associated with the execution will eventually run out. Although EVM bytecode is Turing complete, this limitation creates disincentives for creating sophisticated and resource-intensive smart contracts.

Using Solidity, the procurement legal contract from Figure 1 can be transformed into a smart contract — the associated interface is shown in Listing 1. The contract allows the seller and buyer to invoke behaviour (such as placing an order, terminating the contract and specifying that a delivery was made).

We will now highlight Solidity's salient features required to appreciate this work. Solidity allows standard enumerated types (e.g. line 2 in Listing 1), and key-value associative arrays or mappings (e.g. line 4 defines a mapping from a 16-bit unsigned integer to an `Order` structure used to map from the order number to information, and line 7 of Listing 2 shows how the values can be accessed).

Functions can be defined as (i) `private`: can only be accessed by the smart contract itself; (ii) `internal`: the contract itself and any contract extending it

Listing 1. The interface of a smart contract regulating procurement in Solidity.

```
contract ProcurementContract {
  enum ContractStatus {Open, Closed}
  ContractStatus public status;
  mapping (uint16 => Order) public orders;
  ...

  function ProcurementContract(
     uint _endDate, uint _price,
     uint _minimumItems, uint _maximumItems
  ) public { ... }

  function acceptProcurementContract() public { ... }

  function placeOrder(
     uint16 _orderNumber, uint _itemsOrdered,
     uint _timeOfDelivery
  ) public { ... }

  function deliveryMade(
     uint16 _orderNumber
  ) public byBuyer { ... }

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

can access the function; (iii) `external`: can be accessed from an external call; and (iv) `public`: can be called from anywhere. These access modifiers only define from where a function can be called but not who can call such functions — a `public` function could be called from anyone. As part of a contract it is important to define which parties can initiate different contract functions.

Listing 2. Part of the implementation of the procurement smart contract.

```
1   modifier byBuyer {
2     require(msg.sender==buyer);
3     _;
4   }
5
6   function deliveryMade(uint16 _orderNumber) public byBuyer {
7     Order memory order = orders[_orderNumber];
8     // Ensure that the order exists and has
9     // not yet been delivered
10    require(
11       order.exists && order.status != OrderStatus.Delivered
12    );
13    // Order state update
14    order.status = OrderStatus.Delivered;
15    // Contract state update
16    if (order.deliveryTimeDue < now) {
17      lateOrdersCount++;
18    } else {
19      inTimeOrdersCount++;
20    }
21    // Sign delivery with the courier service
22    courierContract.call(
23      bytes4(keccak256("sign(uint256)")), buyer
24    );
25    // Pay the seller
26    seller.transfer(order.cost);
27
28    emit DeliveryMade(_orderNumber);
29  }
30
31  event DeliveryMade(uint16 _orderNumber);
```

Function *modifiers* provide a convenient reusable method to define ways of modifying the behaviour of functions in a uniform manner, such as this validation logic. Line 1 in Listing 2 defines a `byBuyer` modifier which checks whether the function invoker, retrieved using `msg.sender`, is indeed the `buyer` (the buyer's address would have had to be specified somewhere else in the contract), with the underscore indicating the execution of the original code of the function being

affected by this modifier. Solidity provides such language construct validation guards including `require` which allows for testing of conditions in which case if the condition does not hold, execution will halt and all state changes will be reverted (this can also be done with the `revert`() instruction). It is worth noting that `revert` bubbles up normal function calls i.e. when a function call results in a revert, the calling function also fails and reverts. The only way to stop such revert cascades is to explicitly invoke the called function of another contract using the low-level `addr.call(. . . )` EVM opcode which calls the function given as parameter of the contract residing at the given address, but which returns a boolean value stating whether the call failed or not. Line 19 triggers a signature on a separate contract with the courier, but avoiding the delivery to fail if the signature does not go through for whatever reason. If the call is to be made to a function from another contract, but within the state space of the current one (i.e. having access to the data and functions of the calling contract), a similar opcode `addr.delegatecall(. . . )` can be used.

The `byBuyer` modifier is used in line 6 to ensure that function `deliveryMade` can only be called by the buyer. Note how the underscore at line 3 specifies that the associated function logic (in this case `deliveryMade`()) should be performed after executing line 2.

Each smart contract inherently is also an Ethereum account, allowing it to hold Ether (Ethereum's cryptocurrency) as well as transfer it to other accounts. Incoming transfers are done with function calls which are tagged as `payable`, which enable the caller to send funds when triggering the function. Outgoing transfers can be done using the `addr.transfer(amount)` function, which sends the specified amount of cryptocurrency to the given address. For example, line 21 in Listing 2 performs a transfer of the amount of `order.cost` from the smart contract's internal account to the `seller` account. Finally, Solidity smart contracts can emit events that may be listened to (asynchronously) by applications off the chain. For example, a mobile application can listen to the event defined on line 26, and triggered on line 23 — thus notifying the seller that the buyer has acknowledged receipt and has affected payment.

3 Runtime Verification of Solidity Smart Contracts

ContractLarva is a runtime verification tool for contracts written in Solidity. It works at the Solidity source level of the smart contract and since once deployed, the code of a smart contract is immutable, it is meant to be applied before deployment. As shown in Figure 2, extra code is instrumented into the smart contract based on a given specification, to add runtime checks ensuring that any violation of the specification is detected and may thus be reacted upon.

The tool takes (i) a smart contract written in Solidity; and (ii) a specification written using an automaton-based formalism based on that used in the Larva runtime verification tool for Java [13], and produces a new smart contract which is functionally identical to the original as long as the specification is not violated, but has additional code to (i) track the behaviour of the smart contract

Fig. 2. Workflow using ContractLarva

with respect to the specification; and (ii) deal with violations as given in the specification.

3.1 Runtime Points-of-Interest in Smart Contracts

In any dynamic analysis technique with temporal specifications that express what should happen and in which order, one key element is the identification of which points during the execution of a smart contract can be captured by the analysis and thus analysed at runtime. These points-of-interest, or events[7] typically require balancing between what is required to specify the correctness of the system, with what can be efficiently extracted. Given that ContractLarva works at the source level, it makes sense to annotate points in the control flow of the smart contract in order to generate events when reached. Also of interest are updates to the global state (variable) which may happen at different control points e.g. the status of the procurement contract (whether it is proposed, active or terminated) is set in different functions, even though one may want to ensure that a terminated contract is never reactivated, thus requiring reasoning about runtime points when the status variable is updated. For this reason, ContractLarva also allows the capture of data-flow points-of-interest. These are the two types of events which can be used in ContractLarva:

1. *Control-flow triggers* which trigger when a function is called or control exits that function: (a) `before:function`, triggers whenever `function` is called and before any of the function's code is executed; and (b) `after:function`, triggers the moment `function` terminates successfully (i.e. not reverted). In both cases, the value of the parameters can be accessed by being specified in the event e.g. `before:deliveryMade(_orderId)`, but may be left out if they are not used.
2. *Data-flow triggers*, trigger when an assignment on a global variable occurs (even if the value of the variable does not change) — `var@(condition)` triggers whenever variable `var` is assigned to (just after the assignment is performed), with the `condition` holding. The condition in variable assignment triggers can refer to the value of variable `var` before assignment using

[7] The choice of the term *event*, frequently used in runtime verification, is unfortunately overloaded with the notion of events in Solidity. In the rest of the paper, we use the term to refer to runtime points-of-interest.

`LARVA_previous_var` e.g. to trigger when the procurement contract status goes from `Closed` to `Open`, one would use the event:

```
status@(
  LARVA_previous_status==ContractStatus.Closed &&
  status==ContractStatus.Open
)
```

It is worth remarking that all events trigger if they happen during an execution which succeeds (that is, not reverted). For instance, the control flow event `before:deliveryMade` would not be triggered when `deliveryMade` is called with an order number which does not exist and thus result in a revert due to a `require` statement. Similarly, if `deliveryMade` is called with insufficient gas to execute successfully, the event would not trigger.

3.2 Specifying Properties

In order to characterise correct and incorrect behaviour, ContractLarva uses automaton-based specifications in the form of dynamic event automata (DEAs) — finite state automata with symbolic state, based on *dynamic automata with timers and events* (DATEs) used for specifications in Larva [13] but lacking timers and quantification, and *quantified event automata* (QEAs) as used in MarQ [29], but lacking quantification.

A DEA consists of a deterministic automaton, listening to contract event triggers. A number of the states are annotated as bad states which, when reached, denote that a violation has occurred, and other annotated as accepting states denoting that when reached, the trace has been accepted and monitoring is no longer required. DEAs thus categorise traces into three sets: rejected traces, accepted ones and others which cannot yet be given a verdict. The automata used are, however, symbolic automata — in that they may use and manipulate monitoring variables. Transitions are annotated by a triple: $e \mid c \mapsto a$, where (i) e is the event which will trigger the transition, (ii) c is a condition over the state of the smart contract and the symbolic monitoring state determining whether the transition is to be taken, and finally (iii) a is an executable action (code) which may have a side-effect on the monitoring state, and which will be executed if the transition is taken. Both condition and action can be left out if the condition is true or no action is to be taken respectively.

For instance, consider clause 6 of the legal contract which states that *"Upon termination of the contract, the seller is guaranteed to have received payment covering the cost of the minimum number of items to be ordered unless less than this amount is delivered, in which case the cost of the undelivered items is not guaranteed."* Figure 3(a) shows how this clause may be implemented. The DEA keeps track of (i) the number of items delivered (in a monitoring variable `delivered`); and (ii) the amount of money transferred to the seller (in the variable `payment`). If the contract is closed and the seller has not yet been sufficiently paid (for the minimum number of items to be ordered less any undelivered items), the

DEA goes to a bad state marked with a cross. On the other hand, if during the lifetime of the contract, the seller has already received payment for the minimum number of items to be ordered, the DEA goes to an accepting state (marked with a checkmark) indicating that the property can no longer be violated. Note that any events happening not matching any outgoing transition of the current state leave the DEA in the same state.

However, runtime verification can be used to go beyond ensuring that the smart contract really enforces the legal contract. For instance, although not part of the legal contract, one may expect that the implementation ensures that once the procurement contract is terminated, it cannot be reactivated, a specification of which written using a DEA is shown in Figure 3(b).

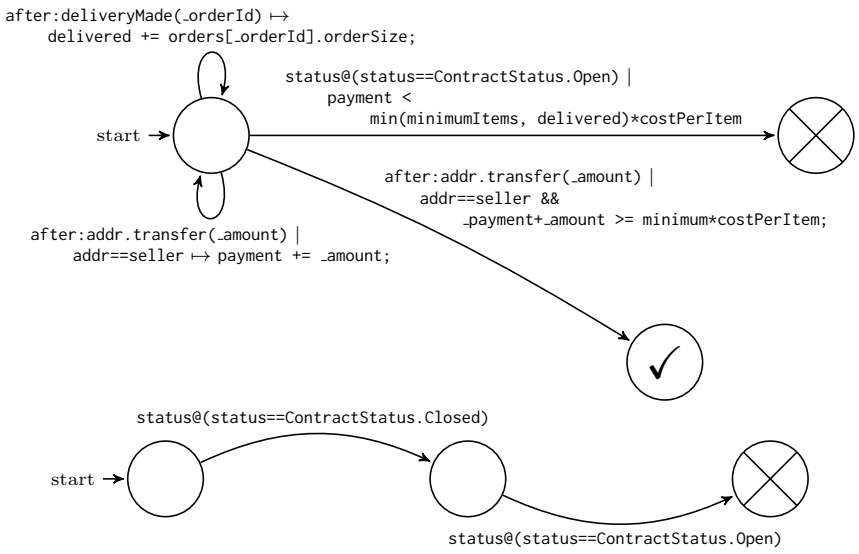

Fig. 3. (a) DEA encoding clause 6 of the procurement contract; and (b) Once terminated, the contract cannot be reactivated.

Formally, DEAs are defined as follows:

Definition 1. *A* dynamic event automaton *(DEA) defined over a set of monitorable events or points-of-interest Σ and system states Ω, is a tuple $\mathcal{M} = \langle Q, \Theta, q_0, \theta_0, B, A, t \rangle$, where (i) Q is a finite set of explicit monitoring states of the DEA; (ii) Θ is a (possibly infinite) set of symbolic monitoring states of $\mathcal{M}$; (iii) $q_0 \in Q$ and $\theta_0 \in \Theta$ are the initial explicit and symbolic state of $\mathcal{M}$; (iv) $B \subseteq Q$ and $A \subseteq Q$ are respectively the* bad *and* accepting *states of the automaton; and (v) $t \subseteq Q \times \Sigma \times (\Theta \times \Omega \to \mathsf{Bool}) \times (\Theta \to \Theta) \times Q$ is the transition relation of $\mathcal{M}$.*

We will write $(q,\theta) \xrightarrow{e,\omega}\!\!\!\!\!\!/\;\;$ *to mean that* $\forall (q,e,c,a,q') \in t \cdot \neg c(\theta,\omega)$.

Informally, $(q,e,c,a,q') \in t$ denotes that if (i) the DEA is in state q; (ii) event e occurs; and (iii) condition c holds, then action a will be executed, updating the monitoring state from θ to $a(\theta)$, and the DEA moves to state q'. Formally the operational semantics are defined as follows:

Definition 2. *The configuration of a smart contract is a pair in* $Q \times \Theta$. *The operational semantics of a DEA* $\mathcal{M}$ *is given by the labelled transition relation over configurations such that* $(q,\theta) \xrightarrow{e,\omega} (q',\theta')$ *holds if and only if, upon receiving event* $e \in \Sigma$ *when the smart contract is in state* $\omega \in \Omega$, *the monitor evolves from explicit state* q *and symbolic state* θ *to* q' *and* θ'*:*

$$\frac{(q,e,c,a,q') \in t \qquad c(\theta,\omega)}{(q,\theta) \xrightarrow{e,\omega} (q',a(\theta))}\; q \notin A \cup B$$

$$\frac{(q,\theta) \xrightarrow{e,\omega}\!\!\!\!\!\!/\;\;}{(q,\theta) \xrightarrow{e,\omega} (q,\theta)} \qquad \frac{q \in A \cup B}{(q,\theta) \xrightarrow{e,\omega} (q,\theta)}$$

The relation is extended over lists of observations, and we write $(q,\theta) \overset{w}{\Rightarrow} (q',\theta')$ *(where* $w \in (\Sigma \times \Omega)^*$*) to denote the smallest relation such that: (i)* $\forall q \in Q,\ \theta \in \Theta \cdot (q,\theta) \overset{\varepsilon}{\Rightarrow} (q,\theta)$*; and (ii) for all* $q,q' \in Q,\ \theta,\theta' \in \Theta,\ e \in \Sigma,\ \omega \in \Omega,\ (q,\theta) \xRightarrow{(e,\omega):w} (q',\theta')$ *if and only if for some* $q_m \in Q$ *and* $\theta_m \in \Theta$, $(q,\theta) \xrightarrow{e,\omega} (q_m,\theta_m)$ *and* $(q_m,\theta_m) \overset{w}{\Rightarrow} (q',\theta')$.

The set of bad *(respectively* accepting*) traces of a DEA* $\mathcal{M}$, *written* $\mathcal{B}(\mathcal{M})$ *(respectively* $\mathcal{A}(\mathcal{M})$*) is the set of traces which lead to a bad (respectively accepting) state:* $\mathcal{B}(\mathcal{M}) \overset{df}{=} \{w \mid (q_0,\theta_0) \overset{w}{\Rightarrow} (q_b,\theta) \wedge q_b \in B\}$ *and* $\mathcal{A}(\mathcal{M}) \overset{df}{=} \{w \mid (q_0,\theta_0) \overset{w}{\Rightarrow} (q_a,\theta) \wedge q_a \in A\}$.

3.3 Reparation Strategies

One of the major challenges with smart contracts is what to do when a violation is detected. Runtime verification on traditional systems typically results in a bug report being filed, and code fixes to be released if the bug is deemed serious enough. In case of the state of the system being compromised due to the issue, the offending actions may be rolled back or manual intervention takes place to ensure correct future performance. In smart contracts reparation to deal with failure which already took place is typically not possible. The (by default) immutable nature of smart contracts means that bug fixes are not necessarily possible and transactions written to the blockchain cannot be easily undone. Immutability comes with caveats (we address this in Section 4) but modification of smart contracts and past transactions goes against the very selling point of using public blockchains: decentralised immutability of smart contracts and transactions.

We typically use dynamic analysis out of necessity when static analysis cannot handle the verification process completely. However in this case dynamic

analysis comes with an advantage: logic to perform actions which override smart contract logic or past transactions can be guaranteed to trigger only when a violation occurs, thus ensuring immutability as long as the code is working as expected. However, the reparation logic itself is typically smart contract- and property-specific. For instance, while a transaction which wrongly disables an order may be fixed by reenabling it, a bug which locks the seller's performance guarantee in the contract (with no means to retrieve the funds) is more complex to address — with one possible reparatory strategy being that of requiring the buyer (or the developer of the contract) to place an amount of funds in the smart contract as a form of insurance, returning them when the contract terminates successfully but passed on to the seller if the performance guarantee becomes locked.

Flexibility of reparation techniques is thus crucial, possibly even more crucial than other domains. ContractLarva allows for custom actions (which may access the system state) which are triggered the moment the DEA moves to a bad or accepting state.

For instance, consider the property ensuring that the procurement contract is not reactivated after being closed. One possible reparation is that of closing it down immediately, which would be handled by the following ContractLarva script:

```
1    violation {
2      contractStatus = ContractStatus.Closed;
3    }
```

This would effectively close the contract immediately to make up for its reactivation. However, the reactivation may happen as part of a more complex transaction (e.g. a function call which, apart from opening the procurement contract, will also make other changes) which one may wish to abort altogether. Using Solidity's notion of reverted computations whose effects are effectively never written on the blockchain, one can build a form of runtime enforcement by effectively suppressing the call which led to the violation in the first place:

```
1    violation {
2      revert();
3    }
```

On the other hand, more complex reparation strategies may require additional code implementing them, as in the case of the minimum order constraint of clause 6 of the legal contract. For instance, the implementation of an insurance-based reparation strategy may work as follows: (i) the party providing insurance must start off by paying a stake before the contract is enabled; (ii) if the specification is violated, the insured party is given that stake; while (iii) if the specification reaches an accepting state, the insurer party gets to take their stake back.

In order to implement this behaviour, the specification would add the following auxiliary code to the original smart contract:

```
1   function payInsurance() payable {
2     require (insuranceStatus == UNPAID);
3     require (msg.value == getInsuranceValue());
4     require (msg.sender == getInsurer());
5
6     insuranceStatus = PAID;
7     LARVA_EnableContract();
8   }
9   function getInsuranceValue() { ... }
10  function getInsurer() { ... }
```

By default, ContractLarva starts off with the original smart contract disabled (i.e. functions automatically revert), and it is up to the monitoring logic to enable it. In this case, the function `payInsurance()` has to be called and the insurance paid by the insurer before the original contract is enabled — `LARVA_EnableContract()` and `LARVA_DisableContract()` are functions provided by ContractLarva to enable and disable the original smart contract. Specification satisfaction (in which case we simply return the stake to the insurer) and violation (in which case the stake is paid to the insured party and the original smart contract is disabled) would then be specified as follows:

```
1     satisfaction {
2       getInsurer().transfer(getInsuranceValue());
3     }
4     violation {
5       LARVA_DisableContract();
6       getInsured().transfer(getInsuranceValue());
7     }
```

For more sophisticated ways of dealing with reparation, including compensations, the reader is referred to [9].

3.4 Instrumentation

Monitor instrumentation into smart contracts can be done in different ways. For instance, instrumentation may be performed at the virtual machine level or at the source code level. It may be achieved by inlining verification code in the smart contract, or by adding only event generation to the original contract, and separate the monitoring and verification code — in the latter case, one may then choose to perform the verification on a separate smart contract or even off-chain. We discuss some of these options in Section 5, and focus on the approach taken by ContractLarva here.

ContractLarva instruments specifications directly into the smart contract at the Solidity source code level, promoting the idea that the new smart contract with instrumented verification code still being accessible at a high level of abstraction. The tool takes a smart contract written in Solidity and a specification, and creates a new smart contract with additional code to handle monitoring and verification.

In order to handle data-flow events, the tool adds setter functions, and replaces all assignments to the monitored variables to use the setter instead[8]. Using these setter functions, instrumenting for data-flow events effectively becomes equivalent to intercepting control-flow events on the setter function.

To instrument control-flow events, we add a modifier for each transition. For a particular event e, we define the set of transitions triggered by it to be $t \restriction e = \{(q, e', c, a, q') \in t \mid e' = e\}$, with which the DEA operational semantics can be encoded. For instance, if $t \restriction$ `before:f(x)` consists of two transitions $(q_1, \texttt{before:f(x)}, c_1, a_1, q_1')$ and $(q_2, \texttt{before:f(x)}, c_2, a_2, q_2')$ we define and use a Solidity modifier to carry out these transitions before `f` is called:

```
1   modifier LARVA_before_f(uint x) {
2     if ((LARVA_STATE == q1) && c1) {
3         LARVA_STATE = q1';
4         a1;
5     } else {
6       if ((LARVA_STATE == q2) && c2) {
7         LARVA_STATE = q2';
8         a2;
9       }
10    }
11    _;
12  }
13
14  function f(uint _value) public LARVA_before_f(_value) { ... }
```

It is worth noting that the overheads induced when a function is called are linear in the number of transitions in the DEA which trigger on events related to that function. In practice, however, one finds that these overheads can be reasonable especially in the context of the critical nature of many smart contracts.

3.5 Runtime Overheads

Although, compared to other verification techniques, runtime verification is typically not that computationally expensive, it performs this computation at runtime, which can affect a program's performance. These runtime overheads can be avoided by performing verification asynchronously, however here we consider synchronous runtime verification since we require monitors to *ensure* that the smart contract conforms to the legal contract.

Unlike traditional systems, where one looks at different dimensions of monitoring overheads: time, memory, communication, etc., in the case of smart contracts on Ethereum, the metric for measuring overheads can be clearly quantified

[8] The only case which is not covered by this approach is if the contract performs external delegate calls (which may result in the callee changing the state of the caller). However, this can be syntactically checked at instrumentation time.

in terms of gas units. The main challenge is that gas is directly paid for in cryptocurrency, meaning that overheads have a direct economic impact[9].

When evaluating instrumented smart contracts we then can first measure the gas cost instrumentation adds to deployment of the smart contract (this additional gas cost reflects the instrumentation logic added), and secondly evaluate function calls to the smart contract to measure increased execution costs. We use this approach to evaluate an application of ContractLarva in the next section.

4 Safe Mutability of Smart Contracts

An important aspect of smart contracts in Ethereum is that they are immutable (once deployed the smart contract's code cannot be changed). This ensures that no one can change the behaviour of the smart contract, protecting users from malicious changes. On the other hand immutability does not ensure this completely, given that smart contracts can call other smart contracts — any change in the target address of such calls changes the control-flow behaviour of the calling smart contract. Previous work shows that at least two out of five smart contracts are not control-flow immutable [16], and thus users cannot be completely sure that the behaviour will not be changed to their detriment and without notice.

4.1 Mutable Smart Contracts

Not allowing such external calls in contracts is not an option, since it is essential to support code reuse and to combine services. Moreover, since smart contracts are programs, they will have bugs, which must be repaired, thus some level of mutability allowing at least bug correction to occur is essential. Here we discuss an approach we proposed in [9] that allows safe mutability of a smart contract through the monitoring of a *behavioural contract*.

As a case study we consider the ERC-20[10] token standard [31]. This standard, which is adhered to by over 100,000 smart contracts[11], is used by smart contracts which implement tokens — virtual assets which may be owned and transferred. Such tokens implement the Ethereum interface shown in Listing 3. Other, less widely used token standards exist, but they all carry out similar functionality and are thus amenable to roughly the same specification we use here.

[9] Although in traditional systems, overheads in space, time and communication are still paid for financially (more memory, more CPU power or more bandwidth), the cost is indirect and the perception is that is a matter of *efficiency*, and not *cost management*.

[10] ERC stands for *Ethereum Request for Comment,* with 20 being the number that was assigned to the request.

[11] As reported by Etherscan (see `www.etherscan.io/tokens`) in July 2018. The number of active, and trustworthy token implementations is, however, much lower than this figure.

Listing 3. ERC-20 token interface standard [31].

```
interface ERC20 {
  function totalSupply() public constant returns (uint);

  function balanceOf(address tokenOwner) public constant
     returns (uint balance);

  function allowance(address tokenOwner, address spender)
     public constant
     returns (uint remaining);

  function transfer(address to, uint tokens) public
     returns (bool success);

  function approve(address spender, uint tokens) public
     returns (bool success);

  function transferFrom(address from, address to, uint tokens)
     public
     returns (bool success);
}
```

An implementation of this standard may allow for possible updates to occur by introducing a *proxy* or *hub-spoke* pattern — a design pattern consisting of a hub (or proxy) contract that serves as the entry-point, which delegates the business logic to another contract. This common pattern allows one to deal with versioning in Ethereum (by allowing the implementation to be dynamically changed simply by updating the address to where the current version of the implementation resides), but does not provide any security to the user, since it allows the owner to change the behaviour unilaterally (e.g. the owner can change the implementation to one that steals commissions from token transfers). To provide the user with more guarantees, we propose the use of behavioural contracts that specify the behaviour the user can expect out when using this smart contract (i.e. the hub), which moreover we monitor for at runtime to revert any illicit behaviour.

As our hub or proxy, we create a smart contract that respects the interface in Listing 3, but which contains no logic except that it passes function calls to the implementation residing in another smart contract which contains the current version of the business logic:

```
  ERC20 implementation;

  function totalSupply() constant returns (uint){
      return implementation.totalSupply();
  }
```

In order to update versions, one can add simple logic to the hub or proxy that allows the owner to update the implementation to one residing at a new address:

```
1  address owner;
2
3  function updateImplementation(address newImplementation)
       public {
4      require(msg.sender == owner);
5      implementation = ERC20(newImplementation);
6  }
```

The ERC-20 standard also comes with behavioural constraints, described informally in [31]. We can specify these using DEAs (see Figures 4, 5 and 6). For specification legibility, we will use the condition denoted by an asterisk (*) to denote an *else* branch for the relevant event i.e. $e \mid * \mapsto a$ will trigger if and only if event e is detected, but no other outgoing transition from the current state is triggered.

In order to ensure that updates to the implementation do not result in spurious, buggy or, even worse, malicious code, we instrument runtime checks to ensure that the effect of the ERC-20 functions on the state of the smart contract are as expected e.g. upon a call to the `transfer` function the balance of the sender and the recipient of the token value is stored, and this is used to check that the exact amount of tokens is transferred appropriately from the sender to the recipient (if the sender has enough tokens).

Thus, by instrumenting the entry-point (or hub) smart contract with this behavioural contract we ensure detection when smart contract mutability results in unexpected or wrong behaviour. If any non-conformant behaviour is detected, a bad state is reached and the transaction is reverted, thus protecting the user from malicious behaviour.

We give some examples of allowed and disallowed traces, using natural numbers as addresses, i.e. `0.transfer(1, 10)` denotes the address 0 calling the transfer function that sends ten tokens to address 1. Consider that `balanceOf(1) == 0` holds then the trace `0.transfer(1, 100);1.transfer(2, 101);` fails, due to Figure 4, while `0.transfer(1, 100);1.transfer(2, 100);` succeeds. For Figure 6, `0.approve(1, 100);1.transferFrom(0, 1, 50);` is successful, but extending it with `1.transferFrom(0, 1, 51);` fails, given that after spending fifty tokens user 1 is only allowed to spend a further fifty tokens.

Note this still has some limitations, namely in terms of securing state, e.g. the owner can still update the implementation that behaviourally respects our contracts but that changes the token values assigned to certain users. To handle this, we can separate the business logic from storage, keeping them in different smart contracts. In this manner, we can allow version updates to the business logic but not to the storage smart contract. In other cases, it may be useful to allow the owner to change the state in special circumstances (e.g. to remedy a mistaken transfer). We do not consider this further here.

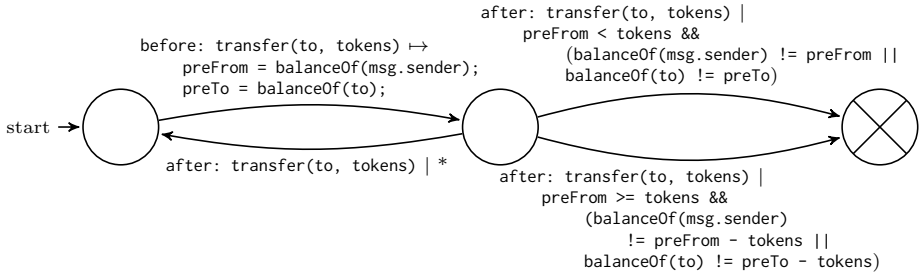

Fig. 4. Calling `transfer` (i) moves the amount requested if there are enough funds; but (ii) has no effect otherwise.

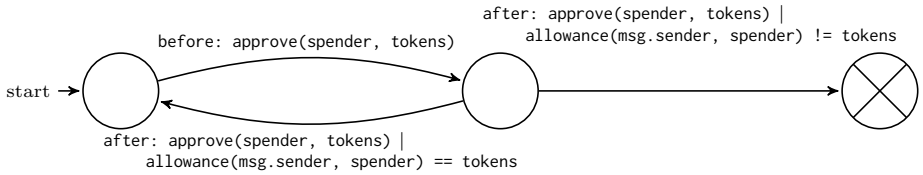

Fig. 5. Calling `approve` changes the allowance to the specified amount.

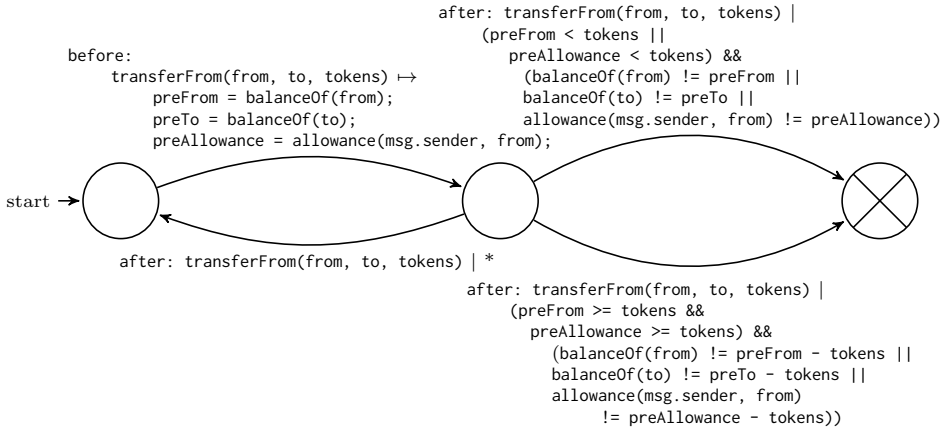

Fig. 6. Calling the `transferFrom` (i) moves the amount requested and reduces the allowance if there are enough funds and the caller has enough of an allowance; but (ii) has no effect otherwise.

4.2 Evaluation

We evaluated the overheads induced by this approach[12] to safe mutability by measuring the associated increase in gas. We measure this in two stages. First we compare the overheads associated with adding versioning (and logic in the spoke to only allow the hub to use the spoke) against the simple case of just using the implementation directly. Secondly we compare the overheads associated with adding monitoring of the behavioural contracts on top of the versioning hub. We also consider some example traces that are

Table 1. Overheads associated with adding a behavioural interface to an ERC20 token.

	Overheads when adding only versioning		Overheads when adding behavioural contracts		Total	
Transactions	Gas Units	Percentage	Gas Units	Percentage	Gas Units	Percentage
Setting up	1711984	65.11%	973794	37.03%	2685778	102.14%
`totalSupply`	4186	18.24%	734	3.2%	4920	21.44%
`balanceOf`	4494	18.71%	734	3.06%	5228	21.77%
`allowance`	4678	18.00%	756	2.91%	5434	20.91%
`transferFrom`	5324	5.78%	93320	101.34%	98644	107.12%
`transfer`	35362	71.47%	76152	153.92%	111514	225.39%
`approve`	5668	8.39%	43462	64.31%	49130	72.70%

The magnitude of these overheads are shown in Table 1, along with the total overheads added when doing both. Note how both introducing a hub-spoke pattern and monitoring introduces substantial overheads. Setting up versioning increases gas costs by up to 65.11%, given the creation of a new smart contract and adding logic to the implementation to only be used by the hub. Moreover, there is a substantial increased cost to using the `transfer` function given it implicitly depends on the `msg.sender` which with versioning must be passed on directly from the hub (otherwise upon a call from the hub the implementation will see the hub's address for `msg.sender`, not the actual transaction initiator's address). When adding monitoring, calling `transfer`, `transferFrom`, and `approve` experience a significant increase in gas costs, which is to be expected given each call to these functions checks at least two monitor transitions. However, it is worth noting that the overhead induced is constant no matter how complex the token logic is. For the sake of this analysis, we used a trivial token implementation, but typically, tokens include more complex (i.e. more expensive) logic, thus reducing the percentage overhead for each call.

Furthermore, the major selling point of smart contracts has been that of guaranteed performance without the need for centralised trust (e.g. in a server), and yet there have been all too many cases of bugs in smart contracts which

[12] The case study can be found at: `https://github.com/shaunazzopardi/safely-mutable-ERC-20-interface`.

result in behaviour which resulted in losses of the equivalent of millions of dollars. Immutability (i.e. non-updatable code) results in bugs and exploits also being immutably present — the guaranteed performance is on the implemented behaviour, though possibly not the originally intended one. Unbridled version updates by the contract owner or developer, result in reintroducing the party who can update the code as a central point of trust, thus questioning the need for a smart contract in the first place. The need for controlled code updates is thus a real one, and the cost can be justified due to the immense potential losses. However, it is still a major question as to how these overheads can be significantly reduced.

5 Open Challenges

In this section we outline a number of research challenges and directions which are still to be addressed for smart contract monitoring.

5.1 Dealing with Failure

In many domains, failure to perform a subtask is handled within the normal execution of the systems, either through return values denoting failure or through exceptions. In either case, the side effects of the computation, both those leading to the failure and its handling can be monitored. In contrast, on Ethereum and Solidity, one can trigger failure through the use of `revert` which rolls back the prefix computation before the failure as though it never happened (apart from the fact that there was a reverted call). Although there has been some related work with runtime monitoring of rollback and compensation computation [10,11], in the context of smart contracts the notion of reverted execution goes beyond simply that of a computation which did not go through. With the view that smart contracts are effectively self-enforced contracts, a legal right such as *'The seller has the right to request an extension of the delivery deadline of an order'* goes beyond having a function `requestDeadlineExtension()`, since if every call to the function by the seller is reverted, the right is not really being respected.

The only way to handle reverted computation on the chain (on Ethereum) is by making the function calls from *another* contract, which allow capturing a `revert` within the logic of the (calling) smart contract, and we have already started experimenting with a variant of ContractLarva which handles an additional event modality `failure` such that the event `failure:f` triggers when function `f` is called but fails due to an explicit `revert` (or instances of the command hidden in syntactic sugar such as in `require`) [9]. If the cause for the failure is lack of gas, however, monitor execution cannot be carried out, which thus leaves the option of violating rights through excessive gas use.

Factoring in gas usage in monitoring for failures is a major challenge. Whether it is through the use of worst-case gas consumption analysis to statically reject monitored functions which may have a gas leak, or whether to leave sufficient gas to deal with monitoring upon a failure, static analysis could support these forms

of violations. Some static analysis techniques to deal with potential gas attacks have already started being investigated [18]. Other options may use dynamic analysis to monitor gas usage for this form of denial-of-right attack.

5.2 Dealing with Monitoring Overheads

Over the past few years, much work has been done applying static analysis to make runtime verification cheaper, including [8,1,14,7]. In the domain of smart contracts, we believe that many of these approaches will perform better, and can be specialised to yield more optimisations. Although smart contract platforms such as Ethereum provide Turing-complete programming capabilities, in practice, few smart contracts use general recursion or loops other than using fixed patterns e.g. iteration through an array. This means that many static analysis techniques, such as abstraction or symbolic execution can yield much more precise results and hence are more effective in reducing runtime verification overheads.

In ContractLarva, we perform all monitoring and verification online and on-chain. Other alternative approaches could include pushing parts of the verification computation off-chain. For instance, for cases where the verification algorithms can be particularly expensive, one may simply log the relevant events (or even use the information written on the blockchain to extract it), and let the parties involved in the smart contract to perform verification, allowing progress only if they agree on the outcome of the verification e.g. using an external oracle, or via a voting mechanism or by all parties having to submit a hashed state of the verifying algorithm. The challenge in designing such an approach is to ensure that a smart contract is not stalled when things are detrimental to some party. Similarly, one may allow for asynchronous monitoring to avoid bottlenecks and enforce synchronisation only when critical situations are reached [12].

Another aspect is that on-chain stateful monitoring is simply impossible on DLTs which have stateless smart contracts, such as Ardor[13]. However, in the case of Ardor, only the relevant parties to a transaction execute the smart contract, and one may consider adding verification modules to clients in order to verify transactions before they are written to the blockchain.

5.3 Beyond Monitoring of Simple Smart Contracts

There are various other open challenges in the field. Our approach focusses on the behaviour of a single smart contract, even though they execute in a context. One may have properties across multiple interacting smart contracts e.g. the procurement smart contract may directly invoke and use a contract with a courier service to deliver the goods. If all the contracts are instrumented with monitoring code, the challenges are similar to those encountered in the monitoring of distributed systems e.g. where there should be a central monitoring orchestrator, or whether monitoring should be split and choreographed across contracts.

[13] See `https://www.ardorplatform.org/`.

If a contract cannot be instrumented with monitoring code, techniques such as assume-guarantee reasoning may need to be adopted to allow for compositional monitoring without being able to monitor within each component.

Although we have focussed on the monitoring of smart contracts, one may look at incorporating monitoring at the level of the DLT itself, beyond the effects of transactions, to include behaviour of miners and the data on the ledger itself. For instance, on Ethereum the order in which transactions are applied and recorded on a new block depends on the miners, which gives rise to a number of vulnerabilities due to a set of miners acting as *malicious schedulers*. Dynamic analysis of miner activity could be investigated to identify such behaviour.

6 Related Work

In this paper we have considered a runtime monitoring approach to the verification of smart contracts, however proving smart contracts safe before deployment is preferable, when possible. Although in their infancy, such approaches to formal verification in Ethereum exist. For instance, the approach proposed in [2] uses deductive analysis to verify business-logic properties of smart contracts at the Solidity level. In contrast, ZEUS [22] uses an abstraction of Solidity code that is translated to LLVM bitcode, allowing for conservative verification of safety properties expressed in a quantifier-free first order logic. This approach however does not soundly abstract all Solidity instructions, given lack of clear counterparts in LLVM, in fact reverting a program state is handled just as a program exit. Moreover external function calls are handled non-deterministically given that the target smart contract of such calls may change at runtime. The same behaviour for external calls is taken by other tools, e.g. [19]. In [19] the sound static analysis tool EtherTrust is used to show that an external call cannot call again the smart contract and reach another external call (then possibly causing an infinite loop that exhausts all gas). Given the external smart contract is not available, this depends on having appropriate logic preventing this in the smart contract. This is a good use case for runtime monitoring tools such as ContractLarva, that can be used to add this safety logic around external calls.

Other work, e.g. [21,28,3], translates EVM bytecode into established languages that amenable to theorem provers, however working at this low-level of bytecode abstracts away some valuable information (e.g. loops). Theorem provers also largely require interaction for full proofs, whereas we are interested in automated verification. Symbolic execution engines also exist for EVM bytecode, that allow for analysis of a smart contract in the context of the rest of the blockchain, e.g. [27,25]. [6] is an example of an approach capable of working at the level of Solidity code, where it translates this to F^* code, making it amenable to the languages typechecking.

All this work has been recent and is not yet mature. Runtime verification, on the other hand, is simpler to implement, and gives precise results, unlike the tools we described whose precision varies. On these tools maturing runtime

verification still has value, where it can be used as the tool of last resort — where other techniques only succeed in proving part of a property safe, runtime verification can be employed to prove the rest of the property, as in [1].

7 Conclusions

We have considered smart contracts and motivated the need for their verification, while illustrating the ContractLarva approach to monitoring Ethereum smart contracts by instrumenting smart contracts with event triggering and monitoring logic. Interestingly, this context allows the blocking of violating behaviour at the level of the smart contracting language, while ContractLarva further allows the specification of further flexible reparation strategies in case of violation. We have applied this approach to limit the mutability of a smart contract's behaviour once it is deployed to the blockchain, allowing updates to its logic while ensuring the behaviour is bounded by an immutable behavioural contract monitor. This allows more dependable services to be provided from the blockchain, and limiting the negative effect of bugs before they are corrected.

There are many open challenges left in smart contract verification. Particularly outstanding is the question of how to handle failure of a transaction. Considering an implementation of a legal contract, a failure of a transaction can have legal implications and verification methods can be used to detect such failures, assign blame, and enforce reparations. The role of off-chain analysis is also discussed, as are avenues for marrying this with on-chain enforcement. Monitoring also presents some challenges given it adds the need for more gas, possibly causing the failure of a transaction due to insufficient gas.

References

1. W. Ahrendt, J. M. Chimento, G. J. Pace, and G. Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design*, 51(1):200–265, 2017.
2. W. Ahrendt, G. J. Pace, and G. Schneider. Smart contracts —a killer application for deductive source code verification. In *Festschrift on the Occasion of Arnd Poetzsch-Heffter's 60th Birthday (ARND'18)*, 2018.
3. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 66–77, New York, NY, USA, 2018. ACM.
4. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *POST*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.
5. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.

6. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts. In *The 11th Workshop on Programming Languages and Analysis for Security (PLAS'16)*, 2016.
7. E. Bodden, L. J. Hendren, and O. Lhoták. A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In *ECOOP'07*, LNCS 4609, 2007.
8. E. Bodden, P. Lam, and L. J. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2010.
9. C. Colombo, J. Ellul, and G. J. Pace. Contracts over smart contracts: Recovering from violations dynamically. In *ISoLA*, Lecture Notes in Computer Science, 2018.
10. C. Colombo and G. J. Pace. Monitor-oriented compensation programming through compensating automata. *ECEASST*, 58, 2013.
11. C. Colombo and G. J. Pace. Comprehensive monitor-oriented compensation programming. In *FESCA*, volume 147 of *EPTCS*, pages 47–61, 2014.
12. C. Colombo, G. J. Pace, and P. Abela. Compensation-aware runtime monitoring. In *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2010.
13. C. Colombo, G. J. Pace, and G. Schneider. Safe runtime verification of real-time properties. In *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009*, pages 103–117, 2009.
14. F. S. de Boer, S. de Gouw, E. B. Johnsen, and P. Y. H. Wong. Run-time checking of data- and protocol-oriented properties of Java programs: an industrial case study. In S. Y. Shin and J. C. Maldonado, editors, *SAC*, pages 1573–1578. ACM, 2013.
15. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy, D. A. Peled, and G. Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
16. M. Fröwis and R. Böhme. In code we trust? In J. Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein, and J. Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 357–372, Cham, 2017. Springer International Publishing.
17. N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.
18. N. Grech, M. Kong, A. Jurisevic, B. Lexi, B. Scholz, and Y. Smaragdakis. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL*, (OOPSLA), 2018.
19. I. Grishchenko, M. Maffei, and C. Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 51–78, Cham, 2018. Springer International Publishing.
20. I. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, Nov. 1989.
21. Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, *Financial Cryptography and Data Security*, pages 520–535, Cham, 2017. Springer International Publishing.
22. S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

23. L. Lessig. *Code 2.0*. CreateSpace, Paramount, CA, 2nd edition, 2009.
24. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
25. L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.
26. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `https://bitcoin.org/bitcoin.pdf`.
27. I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018.
28. D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A Formal Verification Tool for Ethereum VM Bytecode. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, November 2018.
29. G. Reger. An overview of marq. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 498–503, 2016.
30. N. Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, (16), 1996.
31. F. Vogelsteller. ERC-20 Token Standard, 2005. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`.
32. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.