Runtime Verification of Ethereum Smart Contracts

Joshua Ellul*, Gordon Pace†
*†Centre for Distributed Ledger Technologies, University of Malta, Malta
Department of Computer Science, University of Malta, Malta
*joshua.ellul@um.edu.mt, †gordon.pace@um.edu.mt

***Abstract*—The notion of smart contracts in distributed ledger systems have been hailed as a safe way of enforcing contracts between participating parties. However, unlike legal contracts, which talk about ideal behaviour and consequences of not adhering to such behaviour, smart contracts are by their very nature executable code, giving explicit instructions on how to achieve compliance. Executable specification languages, particularly Turing complete ones, are notoriously known for the difficulty of ensuring correctness, and recent incidents which led to huge financial losses due to bugs in smart contracts, have highlighted this issue. In this paper we show how standard techniques from runtime verification can be used in the domain of smart contracts, including a novel stake-based instrumentation technique which ensures that the violating party provides insurance for correct behaviour. The techniques we describe have been partially implemented in a proof-of-concept tool CONTRACTLARVA, which we discuss in this paper.**

***Index Terms*—Distributed ledger technology, Smart contracts, Blockchain, Runtime verification.**

I. INTRODUCTION

Blockchain technology is changing the way in which computer systems can regulate the interaction between real-world parties in a variety of ways. In particular the notion of smart contracts, effectively executable transactions enforced implicitly by certain blockchain architectures themselves, have opened opportunities before impossible without the participation of trusted central authorities or resource managers. The term *contract* has been overloaded, from legal contracts which identify ideal modes of behaviour as agreed between parties (but which may not be adhered to), to programming language contracts which support or enforce specification of expected behaviour of parts of a system (e.g. pre- post-conditions in Eiffel [Mey98] and behavioural interfaces [HLL+12]). Smart contracts have added yet another use of this term, to refer to actual executable code which is agreed upon by the participating parties, and which can be executed. Effectively, smart contracts are executable specifications of the way state will change on the blockchain.

Whether specifications should be executable or not has long been debated in computer science (see [Fuc92] vs. [HJ89]). However, what there is agreement upon, is that an executable specification requires, out of necessity, a description of *how* to achieve a desired state as opposed to simply describing *what* that state should look like — and explaining how to achieve something is more complex, and leaves more room for error than describing what the behaviour should look like.

Thus, a legal contract may specify that *"The service provider is prohibited from keeping records pertaining to the user's behaviour for longer than 1 month,"* but a system designed to satisfy this contractual agreement may do so in a number of possible ways e.g. deleting a user's records immediately, deleting all records on the first day of every month, or deleting records only just before they are one month old. For instance, consider the natural language agreement explaining how a casino owner and players will interact shown in Fig. 1, and the function signatures of a smart contract which concretely implements such a casino, follows:

1) *The casino owner may deposit or withdraw money from the casino's bank, with the bank's balance never falling below zero.*
2) *As long as no game is in progress, the owner of the casino may make available a new game by tossing a coin and hiding its outcome. The owner must also set a participation cost of their choice for the game.*
3) *Clauses 1 and 2 are constrained in that as long as a game is in progress, the bank balance may never be less than the sum of the participation cost of the game and its win-out.*
4) *The win-out for a game is set to be 80% of the participating cost.*
5) *If a game is available, any user may choose to pay the participation fee and guess the outcome of a coin toss to join the game. The game will no longer be available.*
6) *The owner of the casino is obliged to reveal the coin tossed upon creating the game within half an hour of a player participating. If the coin matches the guess, the player's participation fee and the game win-out is to be paid to the player from the casino's bank. Either way, the game then terminates.*
7) *If the casino owner does not adhere to clause 6, the player has the right to declare a default win and be paid the participation fee and the game win-out from the casino's bank. At this stage, the game also terminates.*
8) *No player should be allowed to play more than three games in succession.*

Fig. 1. A legal contract regulating a coin-tossing casino

```
contract Casino {
  private uint bankBalance = 0;

  function Casino() public { ... }

  function depositToBank() public { ... }
  function withdrawFromBank(uint _amount) public { ... }

  function createGame(...) public { ... }

  function placeBet(...) public { ... }
  function resolveBet(...) public { ... }
  function timeoutBet(...) public { ... }
}
```

Consider clause 3 of the legal contract: *"as long as a game is in progress, the bank balance may never be less than the sum of the participation cost of the game and its win-out."* The smart contract may implement this in a number of ways, for instance by stopping the casino owner from withdrawing too much money when a game is in progress, or by altogether stopping the owner from withdrawing money as long as a game is in progress. From the player's perspective, the manner in which this is achieved is not important as long as the code really ensures that there will remain enough money to pay

out in case of a player win. Ideally, the correctness of smart contracts is verified statically at compile time, but using automated static analysis techniques to prove general properties of smart contracts has had limited success and until static analysis techniques and tooling for smart contracts develops to provide a means of resolving application specific potential bugs at compile time, runtime verification can provide an interim solution.

This brings to the front one important issue with smart contracts: indeed, smart contracts do exactly what they say they do, but that might not be what was thought the contract would do. Especially as smart contracts grow in complexity, this issue becomes more important. Whether a contract is written by one of the parties participating in a transaction, or by an outsider, participating parties may rightfully fear that there might be obscure ways in which others can exploit the contract to their benefit. There have been well-known instances of such smart contracts, for instance, on Ethereum [ABC17].

The problem boils down to one of program verification. In order to address such concerns, one would need a more abstract, less exploitable way in which one specifies what the smart contract will or will not do — effectively writing a (non-executable) specification. For instance, one might want to ensure that, halfway through a multi-step transfer of resources, none of the parties may initiate another transfer from within the same contract (thus causing the first one to fail). Another example can be that of a smart contract which regulates a gambling scenario: the party playing the casino's role cannot reduce the winning stakes after a player has placed a bet. Against such a specification, one can use testing to try to identify potential bugs (or workarounds) or use formal static analysis techniques. The former approach suffers from the fact that it lacks complete coverage (there may lie undiscovered bugs in execution paths which were not explored) while the latter typically fails to scale up as smart contracts increase in size and complexity.

In this paper, we propose to use techniques from runtime verification [LS09] to ensure that all execution paths followed at runtime satisfy the required specification, embodied in a prototype tool we have built, namely CONTRACTLARVA[1]. One of the challenges lies in what to do if a violation does occur. In our approach we support different ways to react to violations. At a simplest level, one can block the smart contract from executing further (other than emptying its content as specified in the property). However, we also support an approach based on a stake-placing strategy in which any party that can potentially violate the contract, pays in a stake before running the contract, which will be given to aggrieved parties in case of a violation, but returned to the original owner if the contract terminates without violations.

This proposed violation resolution procedure can be applied in a variety of settings, from when the developer of the contract is one of the parties, and thus the other parties may request guarantees about their behaviour (possibly through a negotiation phase), to when the developer may be a paid third party, who may be asked to guarantee certain behaviour of the contract.

[1]The name is inspired due to the automaton-based approach adopted as used in Larva [CPS09], a runtime verification tool.

Fig. 2. Workflow using CONTRACTLARVA

II. A FRAMEWORK FOR SAFE SMART CONTRACTS

In the runtime verification framework we are adopting (see Fig. 2), we enable the combination of a smart contract and its specification. These are automatically transformed into a safe contract which behaves just like the original one but, in addition, can identify when the specification is violated and trigger remedial behaviour. The framework allows for disabling a smart contract upon the identification of a violation, but it also takes advantage of the application domain, that of smart contracts, to provide stake-based correctness guarantees, in which a party is prepared to provide insurance that the given contract satisfies a particular property. If the property is violated, the violating party will pay the aggrieved party the agreed upon insurance, ensured as part of the automatically generated safe contract, all done in a decentralised manner.

The basic requirements of our framework are a decentralised resource management system and smart contracts which can be written in a language expressive enough to match a trace to a given specification. A proof-of-concept version of the framework has been implemented for Ethereum, transforming smart contracts written in Solidity into safe ones. The Turing completeness of Ethereum smart contracts allows the implementation of a compliance engine which uses Ether for guarantees.

A. Runtime Violation Reparation

Given that in runtime verification, violations of a property are only discovered at runtime, what should be done (and indeed what can be done) is debatable. Various strategies have been proposed and adopted in the literature, ranging from simply logging the violation or stopping the system from advancing further, to more proactive approaches such as enforcing behaviour [Fal10] or compensating for the unexpected behaviour. [CP14]. In general, however, most runtime verification systems simply allow the specification engineer to specify code which will be executed upon violation.

In the context of contracts, however, the particular domain is more specific than that of general computer systems. Firstly, we have the notion of parties participating in a contract. This means that many properties can be associated with (i) a party who can be held responsible for its violation; and (ii) a party (or multiple ones) which can be identified as the aggrieved

party in case of a violation. In addition, contracts typically identify reparatory behaviour which will be enforced in case of non-compliance [PS09].

In the case of smart contracts, we lie somewhere in between the two domains. We have both executable code and a contract over behaviour which is enforced, but with no means of enforcing external behaviour as can be done in legal contracts. For this reason, in our approach we support violation handling by allowing the parties to specify reparation actions written as code as part of the specification given to CONTRACTLARVA. However, we have further developed a contract design pattern which can be automatically engineered to allow parties to place stakes as a guarantee in the case of contract property violation.

Using our approach, contracts can be extended to offer monetary reparation in case of violation. By associating a contract property with (i) the party taking responsibility in case of a violation; (ii) the aggrieved party or parties in case of a violation; and (iii) the amount placed as a guarantee against violation, our tool automatically weaves code into the contract to ensure for each property that (i) the responsible party must initially pay the stake corresponding to the reparation; (ii) if the property is violated, then the aggrieved party receives the reparation stake; and (iii) if the contract will be destroyed without having violated the property, then the responsible party gets back his stake.

The approach can be used even when the specification includes constraints which may not be enforced by the smart contract, but ones which the parties guarantee to each other. For instance, a smart contract may allow to increase a wager any number of times, but one of the parties is willing to guarantee that she will not change the wager from her end more than three times, with a payment penalty in case she does not stick to this constraint. Interestingly, this allows for mutual guarantees over a single smart contract e.g. one party promising to not to change the wager more than three times, while the other guarantees not to wager more than a certain amount. This is more akin to a legal contract in which the parties may violate the constraints, but will have to pay a penalty if they do so.

B. Contract Properties

In order to specify properties, we adopt an automaton-based approach, effectively a subset of the DATE (Dynamic Automata with Timers and Events) as used in the runtime verification tool Larva [CPS09], but without timers.

Such specifications monitor for events[2] over the contract and enable the specification of event traces which are not desirable. The choice of which events are monitorable greatly influences the overheads induced (for example, capturing variable change events can be costly if the variable is frequently updated), but also affects the expressiveness of the specification language. In CONTRACTLARVA, we capture two types of events: (i) control-flow events corresponding to entry and exit points of functions defined in contracts, written $f^{\downarrow}$ and $f^{\uparrow}$ to refer to the entry and exit point of function f respectively; and (ii) data-flow events corresponding to changes in values of variables, written as $v@e$ to denote the event when variable v is changed and expression e (which can also refer to the previous value of v as $\overleftarrow{v}$) holds e.g. $winout@(winout < \overleftarrow{winout})$ identifies points in the execution of the contract when the win-out amount is decreased.

[2]The choice of the term *event* is unfortunately overloaded with the notion of events in Solidity. In this paper, the use of the term is limited to the notion of event triggers as used in DATEs unless explicitly otherwise noted.

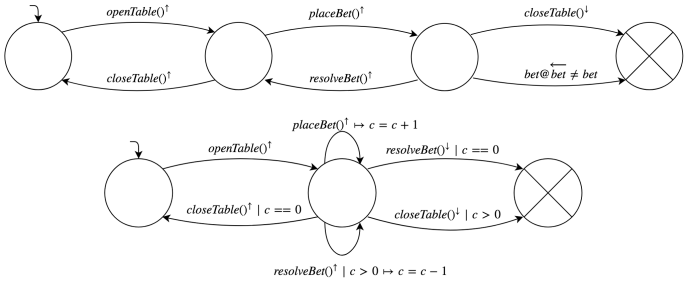

Fig. 3. Contract specification examples (a) a betting table may not be closed when there is a placed wager which has not been resolved; (b) generalisation of the previous specification to allow for multiple simultaneous wages to be placed on the table.

At their most basic level, our specifications will be expressed as (deterministic) automata, listening to contract events. States annotated with a cross denote that a violation has occurred. For instance, consider a smart contract which allows for its initiator to open the gambling table (*openTable*), on which other users may place a wager (*placeWager*), and then resolve it (*resolveWager*) any number of times. The table creator may close down a table (*closeTable*) as long as it has no unresolved wagers. The automaton shown in Fig. 3(a) ensures that a violation is identified if the table is closed when a wager has been placed but not resolved.

The automata used are, however, symbolic automata — in that they may use and manipulate variables. Transitions are annotated by a triple: $e \mid c \mapsto a$, where e is the event which will trigger the transition, c is a condition over the state of the contract (and additional contract property variables) determining whether the transition is to be taken, and finally a is an executable action (code) which will be executed if the transition is taken. Fig. 3(b) is a generalisation of the previous property, to handle the case when multiple wagers may be simultaneously placed and resolved on the same table. The actions typically impact just a number of variables local to the monitors (i.e. not the state of the system itself), although in some cases, however, specifically in the case of a property violation, one may choose to change the system state in order to make up for violated invariants.

For the formal semantics of this notation, the interested reader is referred to [CPS09].

C. Instrumenting the Monitors

One consideration in the development of a runtime verification tool is that of how the monitors will be instrumented.

Although the consideration is internal in that it is invisible to the user, the choice of instrumentation policy can have a direct affect on performance. We have identified two main approaches which can be used to instrument monitors in a smart contract setting:

a) Monitoring as a separate contract: One way of instrumenting the monitor is to translate the specification into a separate smart contract, which is sent the relevant DATE event triggers from the monitored contract. The monitoring smart contract provides all the functionality to keep track of the state of the DATE and updating it upon receiving a DATE event from the original contract. This approach allows the monitoring of multiple smart contracts against a common specification, with the monitor effectively acting as an orchestrator. The major challenge is that the separate monitor does not necessarily have access to the state of the original smart contract, and thus, information about the condition on a transition has to be passed with the event trigger itself, and actions may have to trigger functions in the original contract.

b) Inlined monitors: Another way of instrumenting the monitor, is to instrument the code directly within the monitored smart contract, adding functions implementing the logic required to manage the configuration of the DATE within the monitored smart contract itself. This ensures that the monitor has full access to the local state, simplifying the logic implementing the semantics of DATEs. An underlying assumption with this approach is that the specification sees the smart contract as a monolithic, stand-alone one, and specifications do not span over the behaviour of different contracts.

In the current version of CONTRACTLARVA, we have adopted the latter approach. Solidity source code of the smart contract is parsed, to which monitoring logic is added. Code is added to follow the logic of a DATE by implementing an encoding of the configuration of the DATE, and providing a function *registerEvent* to update the configuration upon receiving a particular event. Solidity modifiers are used to instrument actual invocations of this function in the main code of the original smart contract. In order to deal with data flow events, the contract is updated such that all variables which appear in a $v@e$ event are updated in order to enforce the use of a setter function. Using this approach, $v@e$ events correspond to interception of calls to the setter function.

Upon reaching a violation or accepting state, action is taken, depending on the violating-handling strategy adopted. CONTRACTLARVA currently provides three violation-handling strategies: (i) *stop-upon-violation*, which disables smart contract behaviour once a violation is discovered; (ii) *insurance-against-violation*, which adopts an escrow approach, initially requires the responsible party to transfer into the contract an amount specified within the specification, transferring it back upon reaching an acceptance state but transferring it to the aggrieved party if a violation is detected and blocking all further behaviour; (iii) *multiple-insurance-against-violation* acts just as in the previous case, but allows for the insurance stake to be paid once again to re-enable the smart contract after a violation. To ensure full flexibility, in all cases, we let payout from the contract upon violation to be managed via events in the DATEs.

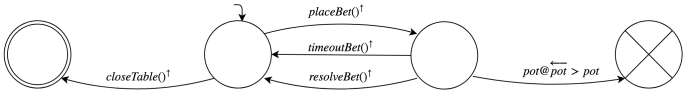

Fig. 4. Property which should be satisfied by the casino contract

It is worth noting that the code generated is a direct implementation of the operational semantics of the subset of DATEs used in the tool, thus ensuring that the verification algorithm is correct. The lack of formal semantics for Solidity impede, however, proving the correctness of the implementation of this algorithm.

III. ILLUSTRATING THE FRAMEWORK

To illustrate the use of runtime verification on a smart contract, we consider a smart contract for gambling on a coin toss. The party setting up the contract takes the role of the casino, and the smart contract allows them to manage their reserves which will be used to pay out winners — allowing the casino to add and withdraw payment from the pot. The casino can also set up a game by sending an encrypted version of the coin result (encoded by hashing an odd nonce to denote heads, or an even one to denote tails) which a player may guess.

Once a game is set up, any other party may bet by depositing an amount (which may not exceed the casino's pot) and a guess. The casino can then decide the outcome of the bet by sending the original number to be verified against the guess. The player may request a default win if the casino does not deposit the original number within a specified time limit. The casino may not initiate more than one game at a time, and only one player can participate in a game. Throughout the process, the casino may always deposit money into the pot, but may only withdraw from it if there is no unresolved player bet.

Consider a specification which says that: *As long as a casino contract is active, the casino party may not withdraw from the pot from the moment a player has placed a bet till when it is resolved.* This can be encoded as a DATE as shown in Fig. 4.

In order to show how monitoring can help, our implementation of depositing to the casino pot does not check for potential overflow. Furthermore, the property is tagged to use one-off stake-based violation handling, with the insurance provider being the party representing the casino (who would have invoked the constructor), and the aggrieved party in case of a violation being the currently active player. Using CONTRACTLARVA with the smart contract and the *Casino* smart contract, we get an new *SafeCasino* smart contract with the specification woven into the original code.

SafeCasino is almost identical to the original *Casino* except that: (i) the *SafeContract* starts up the safe contract in a state awaiting the insurance stake to be paid, until which all functionality is disabled; (ii) a *payStake* function is added to allow the owner (the creator of *SafeContract*) to pay the required stake and enable the rest of the smart contract; (iii) the

current state of the property DATE is encoded in *SafeCasino*, and functions are added to initialise the state, and to update it when an event is received; (iv) functions to handle the situation when the property is satisfied (when an acceptance state is reached, and the stake is paid back to its owner), or a violation is identified (when a bad state is reached, the stake is paid out to the player and all functions in the smart contract are disabled); (v) a setter for variable *pot* (since it is the only variable changes over which may trigger the property transitions) to ensure that any update to the variable can trigger monitoring; (vi) modifiers are created for all functions to ensure that (a) the stake has been paid before proceeding, and (b) to trigger any transitions. For example, a modifier for *placeBet* is created, requiring the monitoring mode to be active and a call is made to follow transitions whenever the function is called, and before it is executed. This modifier is added as the first one to the *placeBet* function:

```
modifier larva_aux_modifier_placeBet() {
     require (larva_mode == LarvaMode.ACTIVE);
     larva_DATE_transition(0);
     _;
}
function placeBet()
  larva_aux_modifier_placeBet . . .
```

Although the function *withdrawFromPot* checks the state of the smart contract and does not allow for withdrawal during a bet, we injected a bug in *depositToPot* (which the casino can use to pay into the pot, and which may be invoked at any time) by not checking for potential overflow. This may lead to a payment into the pot and resulting in an overflow, which will inadvertently decrease the pot. *SafeContract* will immediately identify this violation, pay the player the stake as compensation, and disable the contract. Furthermore, we can specify the violation state to return the original bet to the player and the rest to the casino.

IV. Evaluating the Framework

Evaluating the overheads of a runtime verification tool is very domain dependent. In the case of smart contracts, the major metric is that of increased gas consumption due to the additional monitoring code — a function of code size, memory and execution time. Objective evaluation of a tool's performance is furthermore challenging since these depend on the events and properties being monitored. For instance, low-level properties, typically about events happening with high-density result in higher overheads than business-level properties which work on sparse events. Similarly, properties which have a simple state to keep track of (e.g. pure control-flow properties such as *'the contract cannot be initialised more than once'*) yield lower overheads than ones which require a complex monitoring state (e.g. a property which may require keeping track of the users of a smart contract).

We have performed initial experiments with a real-life smart contract, measuring the impact of monitoring on gas consumption. We have used the Parity Multisig Wallet smart contract [Tec17], looking at version 1.5 of the contract, which included a vulnerability which led to the loss of 30 million US dollars. It is worth noting that the smart contract allows for multiple wallet owners (up to a maximum number), with a specific number of required agreements from the owners to allow a transfer from the wallet.

We have added two types of properties: (i) simple state properties which use only the explicit state of the DEAs to keep track of the runtime behaviour (e.g. *a wallet may not be initialised more than once*); and (ii) additional state properties, which require additional monitoring data structures and code to keep track of the monitoring state through the use of the conditions and actions on DEA transitions (e.g. *the number of required agreements may never exceed the number of owners of a wallet*[3]). Based on these properties, we have made a number of observations: (i) Initialisation is, by far, where the major overhead appears — the increased size of the smart contract and the initialisation costs result in a tenfold execution cost (1100% gas consumption overhead[4]); (ii) Calls to the smart contract which are not involved in the properties yield minimal overhead ($< 1\%$ overhead); (iii) Calls which affect only simple-state or stateless properties result in low overheads ($< 5\%$ overhead); (iv) Calls which affect a complex state obviously depend on the computation required, but can be substantially higher — for instance, keeping track of the owners of a wallet (i.e. duplicating part of the functionality of the underlying smart contract) yielded 20% overhead. Though performance penalties are introduced (much of which occur at deployment time), we believe that the overall run-time overheads justify the protection gained against potential losses (as seen by infamous bugs can run into the millions).

The initial cost may seem over the top, but one has to keep in mind that the contract we are monitoring is a relatively small one (1k LOC, including comments), and thus, the code and state added for monitoring can be proportionately high. The high initial overheads result in the monitoring memory requirements and increase in program size, and are partly due to our choice to use DEAs as a specification language, since much of the memory overheads are statically sized and created upon initialisation. Since this is a one-off cost, and dealt with by the creator of the contract (e.g. the person offering a service), we believe that this is a reasonable choice especially since it avoids extra overheads for others using the contract, and is thus not necessarily a show-stopper. The low overheads for simple-state properties indicates that runtime verification of smart contracts can be viable in practice. On the other hand, one has to be careful of additional symbolic state

[3]To check this property we keep track of the owners of a wallet, requiring additional data and code. We could have also chosen to trust the underlying contract which already keeps track of this, but (i) it is safer to have a monitor with possibly similar logic (but simpler since it keeps track for the sole purpose of verification); and (ii) for some properties, we may not have the luxury of reusing the system state.

[4]Storage on the blockchain is accepted to be one of the most costly operations. Given the relatively small size of the original contract, the monitoring code and memory requirements is substantial, which explains the large overheads for this use case. However, note that (i) this does not grow proportionately as the smart contract grows; and (ii) this one-off cost is amortized over the lifetime of the contract thus making it less of an issue.

which can increase overheads substantially. One advantage of our notation, as borrowed from the runtime verification tool Larva [CPS09], is that the dividing line between what can be efficiently monitored and what can result in more substantial overheads is explicit in the specification notation. We are currently running additional use cases to evaluate the use of ContractLarva in a more thorough manner.

V. Related Work

Despite the extensive literature on runtime verification of computer systems and the use of contract logics for monitoring compliance, there are currently no tools to enable runtime verification of smart contracts. The closest work in the literature uses runtime verification to generate smart contracts which monitor behaviour. For instance, for there is recent work to do this to support business process monitoring [GPDW17], [WXR$^+$16], [PSHW17], where the recurring theme being that of introducing a decentralised monitoring of independent parties by using blockchains to shift trust away from a single central trusted participant orchestrating the monitoring. Both [GPDW17] and [WXR$^+$16] use smart contracts to encode the monitoring of the business processes, while [PSHW17] use the Bitcoin blockchain encoding the monitoring in terms of resource flow instead, due to the lack of smart contracts. In contrast to our work, these works use blockchain and smart contracts as an enabling and support technology for monitoring, rather than use monitoring to support smart contracts. There is also some work [BDLF$^+$16] using formal static analysis (as opposed to dynamic analysis) to analyse contracts written in a subset of the Solidity language, but the approach fails to scale up to many contracts.

Much of the work that uses contracts (in the wider sense of the word) for monitoring (e.g. [HKZ12], [KPS08], [GR10]) uses contracts as specifications. The inbuilt nature of compliance within smart contracts, which can be seen as a way of regulating behaviour, but in fact do so by enforcing behaviour, simply pushes the question of compliance one step away. The Turing completeness of smart contracts comes at a price — their correctness is uncertain. The most similar to our approach is work such as [FPS09] and [GMS10], in which contract descriptions are verified against properties. In [FPS09], contracts are checked for conflicts, while [GMS10] uses model checking to ensure properties of the contracts. However, in such work, contracts are not executable objects, but rather themselves specifications of ideal behaviour thus making the approaches more akin to specification sanity checking.

VI. Conclusions

In this paper we have presented a runtime verification approach to support dependability and correctness of smart contracts, including a proof-of-concept tool implementation of ContractLarva for Ethereum smart contracts written in Solidity. The approach could easily be used to ensure that smart contracts adhere to a given specification. For instance, in the recent, widely reported case of a bug in a smart contract implementing wallets, and which led to huge financial losses, a specification property which stated that a wallet cannot be initialised more than once, or that the ownership of a wallet never changes once initialised could have identified the violation and stopped the financial losses from occurring[5].

References

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.

[BDLF$^+$16] 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.

[CP14] C. Colombo and G. J. Pace. Comprehensive monitor-oriented compensation programming. In *Proceedings Formal Engineering approaches to Software Components and Architectures, FESCA 2014*, 2014.

[CPS09] C. Colombo, G. J. Pace, and G. Schneider. Safe runtime verification of real-time properties. In *Formal Modeling and Analysis of Timed Systems FORMATS 2009*, 2009.

[Fal10] Y. Falcone. You should better enforce than verify. In *Runtime Verification RV 2010*, 2010.

[FPS09] S. Fenech, G. J. Pace, and G. Schneider. Automatic conflict detection on contracts. In *Theoretical Aspects of Computing ICTAC 2009*, 2009.

[Fuc92] N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[GMS10] D. Gorín, S. Mera, and F. Schapachnik. Model checking legal documents. In *Legal Knowledge and Information Systems - JURIX 2010: The 23rd Annual Conference on Legal Knowledge and Information Systems*, pages 151–154, 2010.

[GPDW17] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber. Optimized execution of business processes on blockchain. In *Business Process Management BPM 2017*, 2017.

[GR10] G. Governatori and A. Rotolo. Norm compliance in business process modeling. In *Semantic Web Rules - International Symposium, RuleML 2010*, pages 194–209, 2010.

[HJ89] I. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, November 1989.

[HKZ12] T. Hvitved, F. Klaedtke, and E. Zalinescu. A trace-based model for multiparty contracts. *J. Log. Algebr. Program.*, 81(2):72–98, 2012.

[HLL$^+$12] J. Hatcliff, G. T. Leavens, K. Rustan M. Leino, P. Müller, and M. J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012.

[KPS08] M. Kyas, C. Prisacariu, and G. Schneider. Run-time monitoring of electronic contracts. In *Automated Technology for Verification and Analysis ATVA 2008*, 2008.

[LS09] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[Mey98] B. Meyer. Design by contract: The eiffel method. In *TOOLS (26)*, page 446. IEEE Computer Society, 1998.

[PS09] G. J. Pace and G. Schneider. Challenges in the specification of full contracts. In *Integrated Formal Methods, 7th International Conference, IFM 2009*, pages 292–306, 2009.

[PSHW17] C. Prybila, S. Schulte, C. Hochreiner, and I. Weber. Runtime verification for business processes utilizing the bitcoin blockchain. *CoRR*, abs/1706.04404, 2017.

[Tec17] Parity Technologies. Parity wallet. https://github.com/paritytech/parity, 2017.

[WXR$^+$16] I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted business process monitoring and execution using blockchain. In *Business Process Management BPM 2016*, 2016.

[5]Needless to say, it is easy to identify properties post-factum.