EthVer: Formal Verification of Randomized Ethereum Smart Contracts

Łukasz Mazurek(✉)

University of Warsaw, Warsaw, Poland
lukasz.mazurek@crypto.edu.pl

Abstract. Despite the great potential capabilities and the mature technological solutions, the smart contracts have never been used at a large scale, one of the reasons being the lack of good methods to verify the correctness and security of the contracts—although the technology itself (e.g. the Ethereum platform) is well studied and secure, the actual smart contracts are human-made and thus inherently error-prone. As a consequence, critical vulnerabilities in the contracts are discovered and exploited every few months. The most prominent example of a buggy contract was the infamous DAO attack—a successful attack on the largest Ethereum contract in June 2016 resulting in $70 mln-worth Ether stolen and the *hard fork* of the Ethereum network (80% of Ethereum users decided to revert the transaction and hence two parallel transaction histories exist from that event).

The main contribution of this work is the automatic method of formal verification of randomized Ethereum smart contracts. We formally define and implement the translation of the contracts into MDP (Markov decision process) formal models which can be verified using the PRISM model checker—a state of the art tool for formal verification of models. As a proof of concept, we use our tool, *EthVer*, to verify two smart contracts from the literature: the *Rock-Paper-Scissors* protocol from *K. Delmolino et al., Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.* and the *Micropay 1* protocol from *R. Pass, a. shelat, Micropayments for decentralized currencies.*

Keywords: Cryptocurrencies · Ethereum · Smart contracts · Verification · Formal methods · Model checking

1 Introduction

The notion of a *smart contract* was first introduced in 1997 by Nick Szabo [29]. The main idea behind smart contracts is that many contractual clauses (such as collateral, bonding, delineation of property rights, etc.) can be embedded in the hardware and software and smart contracts are protocols that can serve as digital agreements between the users of the network—the fulfillment of the agreement is automatically guaranteed by the design of the system instead of some external authority like banks, governments or courts.

M. Bernhard et al. (Eds.): FC 2021 Workshops, LNCS 12676, pp. 364–380, 2021.
https://doi.org/10.1007/978-3-662-63958-0_30

The first practical implementations of smart contracts emerged together with the introduction of *Bitcoin* in 2009 [23], however they gained their popularity five years later when Ethereum [10] was announced—the first fully operational digital platform dedicated particularly for smart contracts, much more convenient to use and with much larger capabilities than Bitcoin.

Despite their huge potential capabilities, smart contracts have never been adopted on a large scale, one of the reasons being the fact that it is difficult to verify the correctness and security of the contract. As a consequence, critical vulnerabilities are discovered and exploited every few months [1–3]. The most prominent example of a buggy contract was the infamous DAO attack [15]—a successful attack on the largest Ethereum contract in June 2016 resulting in $70 mln-worth Ether stolen and the *hard fork* of the Ethereum network (80% of Ethereum users decided to revert the transaction and hence two parallel Ethereum blockchains exist from that event).

Several approaches to verification of smart contracts have been proposed, including the automatic and semi-automatic tools which analyze the contract code and check if it satisfies some set of predefined security properties [22,24,32] or user-defined properties [8,11,18–20]. The other line of work focuses on providing tools to help creating smart contracts that follow some security patterns by design [25,28,31]. All these approaches suffer from the following limitation: they focus only on the security of the contract code without analyzing the outcome of the scenario of its usage. In other words, the traditional verification tools answer the questions: *Is this contract guaranteed to not "crash"? Can it end up in an unwanted state? Will the contract function be always executed till the end?* In contrast, none of the methods answers the question: *What will be the result if I use the contract in the following way?* Furthermore, we are not aware of any solution which verifies the randomized smart contracts and its probabilistic properties, for example: *Are the chances of winning in that lottery indeed equal to 1/2?*

1.1 Our Contribution

The main contribution of this work is an automatic method for verifying randomized protocols built on top of Ethereum smart contracts. We introduce the ETV language which allows to easily create such protocols using the syntax of Solidity (the main contract language of Ethereum). Furthermore, we formally define the translation of such protocols into the Markov decision processes (MDPs) which can be verified for security and correctness using the PRISM model checker—a state of the art tool for formal verification of models. The formal translation is accompanied by the implementation of *EthVer*—a fully operational compiler that translates an ETV program into a MDP in the PRISM syntax. As a proof of concept we use our tool to compile and verify two protocols from the literature: the Rock-Paper-Scissors protocol from *K. Delmolino et al., Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.* [14] and the *Micropay 1* protocol from *R. Pass, a. shelat, Micropayments for decentralized currencies* [26].

What distinguishes our work is that we model not only the smart contract code but also the *scenario* of its execution. The syntax of the contract part of ETV language is a slightly modified syntax of *Solidity* (the programming language of Ethereum smart contracts), while the syntax of the scenario part is very similar to *web3.js* JavaScript library (the library used to execute Ethereum smart contracts). This makes the usage of ETV intuitive to anyone familiar with Solidity and web3.js. We are not aware of any other tool that allows to verify the scenario of smart contract execution in such a way. Also, to the best of our knowledge, our method is the first one that allows to verify the probabilistic properties of randomized smart contracts.

Another novel feature of the ETV language is the abstract construction for cryptographic commitments and digital signatures—the important cryptographic primitives present in many smart contracts. The *EthVer* compiler not only translates these abstract objects into MDP (which allows verification) but also provides the exact implementation of commitments and signatures in Solidity and web3.js. This prevents the user from implementing them by hand, which might me cumbersome and error-prone. We are not aware of any other extension to Solidity which offers such functionality.

The full code of the *EthVer* compiler, as well as the extended version of this paper are available at the *EthVer* project page[1].

1.2 Related Work

Among the existing solutions for verification of smart contracts we can distinguish two main groups which can be summarized as the *verification approach* in which the contract is checked for compliance with some specification or security policy and the *design approach* which simplifies the smart contracts creation process by providing frameworks for their development. Below we analyze the related work falling into these two categories and describe how *EthVer* differs from the existing solutions.

Verification Approach. This group contains static analysis tools for automated bug-finding [22,24,32] that verify the code for satisfying some pre-defined security properties, such as the correct order of transactions, timestamp dependency, prodigality or liveness. The other group of tools [8,18,19] provides semi-automatic methods for proving the contract-specific properties. These tools require some manual interaction from the user, such as specifying the loop invariants in the bytecode. Another work [9] analyzes Ethereum contracts by translating them into a functional language F^*. The language provides verification methods and an interactive proof assistant, however the translation supports only a part of the EVM syntax. Other solutions [12,17] provide dynamic monitoring of the predefined security properties, such as transaction order dependency

[1] https://github.com/lukmaz/ethver.

or *callback free executions*, a lack of which is claimed to be the source of common bugs. Both of these methods provide the defense from only a subset of the possible vulnerabilities in the contracts.

All the solutions mentioned in the previous paragraph are able to analyze only the Ethereum Virtual Machine (EVM) pre-compiled bytecode. The other tool [20] analyzes the high-level Solidity contract code using symbolic model checking for the user-defined *policies*. However the policies are restricted to quantifier-free first-order logic, so this method can only solve the state reachability problem and hence, e.g., cannot verify probabilistic properties. Another interesting approach [11] provides a game-theoretic framework in which the smart contract is translated into a concurrent game and the properties of this game are further analyzed using the novel method of abstraction-refinement. This method offers much lower computational complexity than the exact model checking of the whole model, however it does not provide the exact result of the verification, but only the lower and upper bound.

There are several other tools that provide static analysis for generic properties [4–7,30]. None of them is however accompanied by a scientific paper so the full specifications of the actual verification methods are hard to identify.

Design Approach. One example of a high-level language that impose secure design of the smart contract is Simplicity [25]. It is however a general purpose language for smart contracts with no compiler to the EVM bytecode. Another interesting tool allows exporting the compiled code to the intermediate language WhyML[2] which in turn can be checked for security patterns using the program verification platform Why3 [16]. This tool however does not support the full range of properties to verify, in particular it cannot verify probabilistic properties. A slightly different approach [31] introduces *security patterns*—the best practices that must be met while developing the contract code, such as, e.g., performing calls at the end of a function. This approach however does not allow to specify custom properties to be satisfied by the contract.

1.3 EthVer

The *EthVer* compiler falls somehow between the *verification approach* and *design approach*—it is able to verify the actual code of Ethereum smart contract (and also the scenario of its usage), however it requires the contract code to be written in the ETV language which is a slightly modified version of Solidity. Furthermore it allows to verify any custom property written in a dedicated language, including the probabilistic properties.

To the best of our knowledge, none of the solutions described in this section offers the exact model checking of the *probabilistic* properties of the randomized contract. Moreover, the existing approaches focus on verifying the *contract* without taking into account the pattern of execution of the contract by the users.

[2] https://why3.lri.fr.

Instead, in *EthVer* we verify the *protocol* which consists of the contract and the *scenario* of usage of the contract by the users. Hence, we are able to verify not only the correctness and security of the contract code, but also the *instructions* on how to use the contract.

It is worth noting here, that the two features of *EthVer* described above allows to perform the full formal verification of the *rock-paper-scissors* protocol [14] and the *Micropay 1* protocol [26], which cannot be done in any of the other tools analyzed in this section. We briefly describe this analysis in Sect. 7.

2 Preliminaries

2.1 Ethereum Languages

The actual code of Ethereum smart contracts is written in the machine code of Ethereum Virtual Machine (EVM). However, the platform provides several high level, user-friendly languages to write the code of a contract with the Solidity language being the most popular among them. The syntax of Solidity is based on JavaScript with some extra features added to handle the flow of money and cryptographic operations. Calling a contract function is realized by sending a special transaction to the contract address. There are several convenient GUI tools to deploy and execute smart contracts, such as, e.g., a desktop application Ethereum Wallet or a web application Remix as well as the console client `geth`[3]. Under the hood they all use the JavaScript API with the web3.js library[4] which provides the basic functions to interact with the contract as well as some cryptographic functions widely used in smart contracts (such as hash functions and digital signatures). The main web3.js function to interact with a contract is the `sendTransaction` method which is called on a contract function object and takes as arguments the arguments to the function and the sender address (the `from:` field) as well as the *value* attached to the transaction. The example usage of the `sendTransaction` function is listed below:

```
Bank.deposit.sendTransaction(1,
{from: "0x14723a09acff6d2a60dcdf7aa4aff308fddc160c",
  value: web3.utils.toWei("5", "finney")});
```

Note that the transaction value must be passed as an integer number of *wei* ($1\ wei = 10^{-18}\ ether$), however, the `web3.utils.toWei` function can be used to easily convert from different units like *finney* ($1\ finney = 0.001\ ether$).

2.2 The PRISM Model Checker

PRISM is a probabilistic model checker, a tool originally described in [21]. It is designed for formal modeling and analysis of systems which present random or probabilistic behavior. Many smart contracts fit into this category, so we decided

[3] http://ethereum.org/.
[4] http://web3js.readthedocs.io/.

to use PRISM as the backbone for our formal verification of Ethereum smart contracts.

PRISM supports different types of models, including discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs), probabilistic automata (PAs), probabilistic timed automata (PTAs). In *EthVer* we decided to use Markov decision processes, since they allow non-determinism and hence are the best fit for randomized protocols built on top of smart contracts.

The PRISM model is defined as a set of states and transitions between them. Each transition is represented with a *command* of form

$$[label]\ guards\ \texttt{->}\ updates\texttt{;}$$

where *guards* are the conditions needed to be met in order for the transition to be *enabled*, *updates* represent the probabilistic choices in the algorithm, and *label* is an optional identifier of the transition used for synchronization. The syntax of the updates is as follows:

$$p_1\ \texttt{:}\ update_1\ \texttt{+}\ p_2\ \texttt{:}\ update_2\ \texttt{+}\ \ldots\ \texttt{+}\ p_n\ \texttt{:}\ update_n$$

The updates list reflects the situation when several transitions are possible from the same state and the choice of the actual transition is probabilistic: the i-th transition happens with probability p_i and results in $update_i$ of the model variables.

For more detailed introduction to PRISM please refer to appendix A of the extended version of the paper[5].

2.3 Cryptographic Commitments

A cryptographic commitment scheme is a protocol which consists of two phases: *commit* and *open* (the second phase is also referred to as the *reveal* phase). In the most common implementation during the *commit* phase the user chooses a value r to which they will be committed, chooses a random value s and computes $c = H(r, s)$ where H is a hash function (a collision-resilient function that is hard to invert). Then the user publishes the hash c while keeping r and s secret.

In the *open* phase the user reveals the chosen values r and s and anyone can use c to verify if the author of the commitment didn't change r. The cryptographic commitments are *hiding* and *binding* in the sense that:

- the value of c reveals no information about r,
- once the values of r and c are fixed, it is infeasible to come up with another value of r matching the same c.

[5] Recall that the extended version of the paper as well as the code of *EthVer* and example contracts are available at https://github.com/lukmaz/ethver.

Ethereum provides a convenient way to implement the cryptographic commitments using *SHA-3* function (also known as *keccak256*):

```
hash = web3.utils.sha3(web3.utils.
 toHex(r).substr(2)+web3.utils.toHex(s))
```

Listing 1.1. Computing the commitment in web3.js v1.0.0

```
uint8 r; string s; bytes32 c =
 keccak256(abi.encodePacked(48+r,s))
```

Listing 1.2. Computing the commitment in Solidity v0.5.2

Note the different names of the hash function and the subtle differences in passing the arguments. These differences follow from different APIs used by Solidity and web3.js, but the underlying hash function is the same.

2.4 Digital Signatures

Ethereum supports digital signatures based on elliptic curve cryptography implementing the SECP-256k1 standard as described in [13]. The web3.js library provides two useful functions: `web3.eth.accounts.sign(m, sk)` for signing and `web3.eth.accounts.recover(m, signature)` for recovering the public key of the author of the signature. Solidity provides the function `ecrecover` (hash, v, r, s) which takes the hash of the messages and (v, r, s) values, which are the 3 parts of the `signature`[6]. In the most common scenario the signatures are created off-chain in web3.js and then they are later verified by the contract. Due to different APIs of Solidity and web3.js, a special care is needed for the format of numbers passed to the `sign` and `recover` functions. Listings 1.3 and 1.4 show an example code for signing a message in web3.js and verifying the obtained signature in Solidity[7].

```
r2_ = web3.utils.toHex(r2).substr(2);
concat = c + web3.utils.padLeft(r2_, 2)
  + a.toLowerCase().substr(2);
msg = "msg" + web3.utils.sha3(concat);
s = web3.eth.accounts.sign(msg, privKey);
```

Listing 1.3. Signing a message in web3.js v1.0.0

```
string header = "\x19Ethereum
 Signed Message:\n69msg0x";
bytes data = hexToBytes(keccak256(
 abi.encodePacked(c,r2,a)));
bytes32 msgHash = keccak256(
 abi.encodePacked(header, data));
return ecrecover(msgHash,
 s_v, s_r, s_s) == a));
```

Listing 1.4. Verifying the signature in Solidity v0.5.2

3 Interacting with the Contract

The code of a smart contract does not carry all the information needed for verification. Consider a simple Bank contract written in Solidity:

[6] Note that we don't describe here how to sign messages in Solidity. In fact, Solidity does not provide convenient API for this. The reason is that a private key is required to sign and we rarely want to do this in the contract code, because we do not want to reveal the private keys to public.

[7] This is the actual code of computing and verifying the signature $\sigma = \mathsf{sig}(c, r_2, a)$ from the Micropay 1 protocol (cf. Sect. 7).

```
contract Bank {
  unit balance;

  function deposit() public payable {
    balance = balance + msg.value;}

  function withdraw(uint amount) public {
    if (amount <= balance) {
      balance = balance - amount;
      msg.sender.transfer(amount);}}}
```

Listing 1.5. A simple Bank contract

Is this a secure smart contract? The answer to this question depends on how we want to use the contract and what behavior of the contract is expected. For example, this contract can be considered secure if we want a bank in which anyone can deposit money and then anyone can withdraw it. On the other hand, if we define the security of the bank with the rule that *only the person who has deposited the money can withdraw it*, then of course this contract is not secure.

In order to concretize the requirements for the contract we must formulate the *scenario* and the *properties* which we want to be satisfied. In case of the Bank contract they can be as follows:

Scenario:

- User A deposits 10 *finney*[8].
- User A withdraws 10 *finney*.

Properties:

- User A gets back his deposited 10 *finney*.

Of course this property is not always satisfied, which can be shown using the *counterexample* scenario in which user A deposits 10 *finney* and then user B withdraws the same 10 *finney*. After that the user A no longer can withdraw 10 *finney*, since the contract account is already empty.

Although this scenario of using the Bank contract may look artificial (why not to use the contract in a different way?), in case of many contracts, the scenario of the proper usage is obvious and well defined. Consider, e.g., a simple lottery in which user A bets 10 *finney* and wins 20 *finney* with probability 1/2 (otherwise loses). In such case, the scenario and the properties for user A are as follows:

Scenario:

- User A deposits 10 *finney*.
- User A waits for the result of the lottery.

Properties:

- With probability 1/2: the user A receives the reward of 20 *finney*.
- With probability 1/2: the user A receives nothing.

[8] Recall that 1 *finney* = 0.001 ETH is a denomination of Ether, the currency of Ethereum. For simplicity, we neglect the transaction fees, unless stated otherwise.

4 The ETV Language

To model a contract with its scenario in a verifiable way we introduce the ETV language. The ETV program consists of two parts: the first part is a slightly modified Solidity code of the contract, while the second part represents the scenario using web3.js commands.

4.1 Bounded Integers

The main issue with the verification of smart contracts in PRISM is the usage of *bounded integers* in PRISM. The reason for it is that a new state in the PRISM model is created for every valuation of the variables, and thus increasing the range of variables increases the number of states in the model in the exponential way. On the other hand, in Solidity/web3.js the smallest type for storing integers is `uint8` which is capable of storing numbers from the range $[0, 255]$. Frequently we use such type to store the variables which can have only a small number of different values (e.g., only 0, 1 or 2) and we do not need the whole range of `uint8`.

Because of this limitation we introduce in ETV the *bounded integer* type `uint(N)` which in practice is the main difference between ETV and Solidity/web3.js.

4.2 Communication

A protocol can contain some operations that are performed directly between the parties of the protocol (without interaction with the blockchain), for example exchanging hash values. We define a dedicated *communication* section for such operations in the ETV language. Such approach allows us to properly model the *adversarial* behavior by allowing the malicious party to also execute the commands from the *communication* section.

4.3 Cryptographic Primitives

The other important feature of the ETV language is the abstract syntax for cryptographic primitives, such as hashes, commitments and signatures. Such primitives can be (a) translated into PRISM which allows to verify the properties of the contract and (b) translated into the actual implementation in Solidity and web3.js which reduces the probability that the user implements it incorrectly. The last feature is especially important because the current versions of Ethereum programming languages (Solidity v0.5.2 and web3.js v1.0.0) present large differences in the API for the cryptographic functions and a special care must be taken to make sure that the Solidity part and the web3.js part of the code operates on the same numbers[9].

[9] Examples of the syntax of commitments and signatures in Solidity and web3.js have already been presented in Sect. 2.3 and 2.4.

5 The Compiler

The main practical result of this work is the implementation of *EthVer*—a compiler written in Haskell that takes as the input an ETV file (let us call it `example.etv`) and produces:

- `example.sol`—the contract code in Solidity which can be directly deployed to the Ethereum blockchain,
- `example.scen`—the scenario of the execution of the contract containing the exact JavaScript web3.js commands which can be directly used to execute the contract,
- `example.prism`—the PRISM Markov decision process (MDP), which can be directly used in the PRISM model checker.

While translating ETV code to Solidity and web3.js is straightforward, the translation from ETV to PRISM MDP is highly nontrivial and was the main challenge during creation of *EthVer*. In the next section we describe the core concepts behind this translation and their implementation in *EthVer*.

Furthermore, in the extended version of the paper we formally define the full syntax and the semantics of the ETV language (appendices B and C). In appendix D we formally define the translation from ETV to MDP and prove that it preserves the semantics of ETV.

6 Modeling the Protocol as Markov Decision Process

In this section we present the core concepts of *EthVer*—the way in which we translate an ETV program into a Markov decision process (MDP).

6.1 Modeling the Contract Execution

We model the main part of the contract using 4 PRISM modules[10]: `player0`, `player1`, `contract`, `blockchain`. We show the role of each module on the example of the simple Bank contract (cf. listing 1.5) and the following scenario of its usage:

- User A deposits 10 *finney*.
- User A withdraws 10 *finney*.

We model the honest execution of the scenario with the following commands in the `player0` module:

```
module player0
[broadcast_deposit] (state0 = 1) -> (state0' = 2)
  & (deposit_value0' = 10);
[broadcast_withdraw] (state0 = 2) -> (state0' = 3)
  & (withdraw_amount0' = 10);
endmodule
```

[10] A PRISM model can consists of several modules, each corresponding to a different part of the system and each with a separate set of variables.

Each command sets the value as well as all the arguments of the function call and then triggers the corresponding commands in `contract` and `blockchain` modules using the PRISM synchronization mechanism—the command with a non-empty label (the string in square brackets) can be executed only in parallel with the corresponding function with the same label in other modules (as long as such command exists in other modules).

The actual process of calling the contract function consists of two phases: in the first phase, in parallel to `[broadcast_*]` command from `player0` module, PRISM executes the synchronized command from the `blockchain` module which stores the information that this function call is now in the *broadcast* state. Then at some later point PRISM can take one of the function calls from the *broadcast* state and actually execute the corresponding contract code. This is accomplished by another pair of synchronized commands from `blockchain` and `contract` modules[11].

6.2 Modeling the Adversary

Although the verification of the honest execution of the protocol is important, we frequently face vulnerabilities in the contracts which reveal themselves only when one (or more) of the participants misbehave, i.e., deviate from following the scenario. In order to model the adversarial player, we decided to give them the ability to interact with the contract in an arbitrary way. More concretely, the adversary can **call any function of the contract, in any order, with any arguments, as many times as wanted**. With such definition of the adversary we can model any 2-player contract[12] in one of the 3 following *modes*:

- *honest* mode—honest player 0 vs honest player 1
- *adversarial player 0* mode—malicious player 0 vs honest player 1
- *adversarial player 1* mode—honest player 0 vs malicious player 1

6.3 Modeling the Communication

As it was already discussed in Sect. 4.2, some protocols contain phases in which the players do off-chain computation and exchange the computed numbers without calling the contract. Since these procedures do not involve the actual execution of the contract code, they should not be handled in the same way as the

[11] The same pattern of a two-phase function execution could be accomplished using only the `player0` and `contract` modules, however because of visibility of the variables in PRISM, the `blockchain` module is needed to correctly pass the arguments of the call.

[12] The current version of *EthVer* is limited to 2-players protocols only. However, all the security claims as well as the formal translation defined in appendix D of the extended version of the paper hold also for protocols with larger number of players. Note that although *EthVer* accepts only 2-player protocols, it verifies the contract also against the attacks in which more adversarial players join the protocol at the same time.

contract calls are. On the other hand, we do not want to limit the capabilities of an adversarial player, and hence we need to give the adversary the possibility of performing these procedures at any time, with any arguments (like in case of contract calls). We model every such action as a *communication function* that are called during honest scenario execution and can also be freely called by the adversary.

These communication functions are stored in the separate `communication` section of the ETV code. The syntax of such functions is very similar to the syntax of contract functions, with the only difference that it cannot handle the money transfers. These functions translate to the `communication` module in the PRISM code which can be triggered using the PRISM synchronization mechanism from the player modules in a similar fashion to the contract function (but without going through the *broadcast* state and without involving the `blockchain` module).

6.4 Modeling the Cryptographic Commitments

Recall that in the standard implementation of random commitments (Sect. 2.3) during the *commit* phase two random numbers (r and s) are generated and then they are later revealed during the *open* phase. Since all variables in PRISM are *public*, we cannot just store r and s as PRISM variables, because it will break the *hiding* property of the commitment. It follows from the fact that MDPs are non-deterministic and for MDPs we always compute the maximal (or minimal) probability `Pmax/Pmin`, where the probability is computed over all the random choices, while the maximum (minimum) is taken over all the non-deterministic choices of the model and hence the non-deterministic choices must be done before the random choices.

To best illustrate the problem, consider a simple game in which A creates a commitment by choosing r and s at random and then B tries to guess r before the revealing phase. If we store the value of r in a variable right after the *commit* phase, then the automaton that models B can non-deterministically choose the correct value of r (since now there is no more randomness in the protocol) and win the game. Hence, in order to properly model the real behavior of keeping r secret, we need to not store the final value of r during the *commit* phase and postpone the actual random choice until the *open* phase.

In our implementation each commitment in PRISM can be in one of the following states: *init*, *committed*, or *revealed*. All the commitments start in the *init* state. During the honest scenario, when the player *creates a random commitment*, the appropriate variable switches to the *committed* state, but no actual choice of the value is made. During the *open* phase, the player needs to call a separate `revealCmt` method which performs the actual random choice. After this call, the commitment variable switches from the *committed* state to the *revealed* state.

Using the same mechanism the adversary can either commit to a random value (by switching to the *committed* state and postponing the actual choice until the revealing phase) or he can immediately commit to the value of his choice

by switching directly to the *revealed* state. In both cases he cannot later change the chosen value. This models the real implementation of the commitments in which the chosen value also cannot be changed after the commitment is created.

This approach is realized in *EthVer* by providing the `cmt_uint` type and functions `randomCmt` and `verCmt` which implement directly the described functionality.

6.5 Modeling the Digital Signatures

We introduce a templated type for the signatures: `signature(T1, T2, ...)`. The types `T1`, `T2`, ... are the types of the *fields* of the signature—the values that we want to sign (when we want to sign more than one value, we usually concatenate them before signing). For the signature type we provide two constructions to create and to verify the signature:

```
sigma = sign(c, r2, a);
verSig(verAddress, sigma, (c, r2, a));
```

We model the signatures in PRISM in the following way:

- each signature is initially in the *init* state,
- there is a separate PRISM variable for each *field* of the signature as well as for the address of the author of the signature,
- whenever a signature is created, the fields are assigned with the values being signed (and the author's address); these fields' values cannot be later changed.

The adversary is not allowed to change any field of the existing signature. However to not limit their ability to *interact with the contract in any way at any time*, we allow them to freely create new signatures, i.e., to sign any data at any time with their own key.

6.6 Modeling the Time

Solidity natively supports creating contracts dependent on time using the `now` variable. Moreover, we introduce in ETV the `wait(condition, time)` statement which implements the conditional wait: after reaching that point of the scenario, the party pauses the execution of the protocol until the condition is satisfied or a particular time has passed.

We model the time in PRISM using the `time_elapsed` variable and the synchronized commands labeled `[time_step]`. The `time_elapsed` counter is increased in either of the two cases:

- all the honest parties have finished all of their allowed scenarios steps and are waiting on the `wait` statement,
- the honest party has finished all of their allowed scenarios steps while the adversary decides to not execute any step.

This reflects the assumption that the honest parties always follow the protocol and execute every scenario step within a given time limit while the adversary can interrupt the protocol at any time and refuse to execute a scenario step within the time limit.

7 Case Study: Verification of Two Protocols from the Literature

As a proof of concept we use the *EthVer* compiler to formally verify two protocols from the literature: The Rock-Paper-Scissors protocol from *K. Delmolino et al., Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab.* [14], and The Micropay 1 protocol from *R. Pass, a. shelat, Micropayments for decentralized currencies* [26]. The results of verification of these protocols are broadly described in appendices E and F of the extended version of this paper.

In the first work [14] the authors analyze the actual smart contract for the Rock-Paper-Scissors game which was created by undergrad students during the cryptocurrency lab. The authors point out several typical mistakes that were made during designing this contract and present a few good programming practices to avoid such bugs in the future. Using *EthVer* we were able to automatically find all the contract bugs and fix all of them. The case study is iterative—after each bug fix we rerun the verification and every time the *EthVer* shows us the next bug of the protocol. Using this iterative method we implement 6 bug fixes in total which finally lead us to the correct version of the contract.

It must be stressed, that in the original paper the authors analyze the contract and fix all the bugs *by hand*. In contrast, in our experiment the only manual action is that we rewrite the original contract to the ETV language (which requires only a few minor tweaks) and then *EthVer automatically* finds all the bugs and provides the *counterexample* for each of them which make it easy to fix the bugs.

In the second paper [26] the authors present the Micropay 1 protocol—a smart contract which can serve as a platform for *micropayments*—fast and cheap off-chain transactions which only occasionally require the interaction with the blockchain. In the original version of the paper [26] (published on the *22nd ACM CCS '15 conference*) the authors describe a buggy version of the protocol—the contract is vulnerable to so called *front-running attack*. After the publication we discovered the bug (it was also discovered independently by Joseph Bonneau) and contacted the authors with our findings. As a result, they published the corrected version of the paper [27]. In case of this protocol we also were able to find and fix the bug using *EthVer*. Again, the *EthVer* has found the bug *automatically*, which means that if the authors verified the protocol before publication using *EthVer* or a similar tool, they would discover the attack and the buggy version of the contract would never be published.

Our case study involved 9 tested models in total (7 versions of the RPS contract and 2 versions of the Micropay contract[13]). The Table 1 shows the performance of all the test runs. Each test was performed on a laptop with Intel Core i7-4750HQ CPU @ 2.00 GHz and 8 GB RAM.

[13] The ETV code of all tested models is available in the project repository, https://github.com/lukmaz/ethver.

Table 1. Performance of all the test runs

Protocol	Number of states	Model checking time
rps v1	1.6M	130 s
rps v1a	1.2M	72 s
rps v1b	1.9M	75 s
rps v2	0.8M	66 s
rps v3	6.6M	470 s
rps v3a	5.3M	264 s
rps v4	5.2M	238 s
micropay v1	16M	24 min
micropay v2	490M	124 min

8 Conclusions

In this work we present the *EthVer* compiler—a novel tool for formal verification of Ethereum smart contracts. We have developed a dedicated ETV language for designing secure and verifiable contracts. We have formally defined and proved the correctness of the translation of this language to Markov decision process (MDP) models in PRISM. This translation has been implemented in Haskell and works as a standalone computer program.

The novelty of our approach lies in the three features: (1) the verification of the whole cryptographic protocol consisting of a smart contract and a scenario of its execution, (2) the verification of probabilistic properties of randomized contracts, and (3) the abstract language construction for cryptographic commitments and signatures, which can be automatically translated into the actual Ethereum code and into the PRISM model. To the best of our knowledge, no other verification approach offers any of these 3 functionalities.

The automatic verification of the model generated by *EthVer* is possible due to our original method of modeling the contract as MDP. Our technique allows to verify the correctness and security of the honest execution of the protocol and also verifies the protocol against the attacks of the adversarial user. Moreover, in case the vulnerability of the protocol is found, our tool returns the *counterexample*—the execution path which leads to the undesired state of the protocol.

As a proof of concept we used *EthVer* to verify two smart contracts from the literature. In both cases *EthVer* was able to automatically find the bugs that were claimed to be found manually by the authors. This means that the vulnerable contracts would not have been created if the authors had used *EthVer* for their verification.

The experiments results show that the verification is practical—it can be performed on a medium-class PC within a reasonable time frame. However, the experiments revealed also the inherent limitation of our method—the size of the model (and hence the verification time) grows exponentially with the number of parameters of the contract. Therefore our method is most suitable for the

contracts of a limited size—for larger models the exact model checking is not possible and other verification methods must be used.

References

1. Accidental bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html. Accessed 2 Mar 2019
2. How to find $10m just by reading the blockchain. https://coinspectator.com/news/539/how-to-find-10m-just-by-reading-the-blockchain. Accessed 2 Mar 2019
3. An in-depth look at the parity multisig bug. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/. Accessed 2 Mar 2019
4. Manticore. https://github.com/trailofbits/manticore
5. Mythril. https://github.com/ConsenSys/mythril
6. Smartcheck. https://github.com/smartdec/smartcheck
7. solgraph. https://github.com/raineorshine/solgraph
8. Amani, S., Bégel, M., Bortin, M., Staples, M.: 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, pp. 66–77. ACM, New York (2018). https://doi.org/10.1145/3167084
9. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
10. Buterin, V.: Ethereum: a next-generation smart contract and decentralized application platform (2014). https://github.com/ethereum/wiki/wiki/White-Paper. Accessed 22 Aug 2016
11. Chatterjee, K., Goharshady, A.K., Velner, Y.: Quantitative analysis of smart contracts. In: Ahmed, A. (ed.) ESOP 2018. LNCS, vol. 10801, pp. 739–767. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89884-1_26
12. Cook, T., Latham, A., Lee, J.H.: Dappguard: active monitoring and defense for solidity smart contracts. Retrieved July 18, 2018 (2017)
13. Courtois, N.T., Grajek, M., Naik, R.: Optimizing SHA256 in bitcoin mining. In: Kotulski, Z., Księżopolski, B., Mazur, K. (eds.) CSS 2014. CCIS, vol. 448, pp. 131–144. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44893-9_12
14. Delmolino, K., Arnett, M., Kosba, A., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: lessons and insights from a cryptocurrency lab. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D., Brenner, M., Rohloff, K. (eds.) FC 2016. LNCS, vol. 9604, pp. 79–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_6
15. Falkon, S.: The story of the DAO – its history and consequences (2017). https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee
16. Filliâtre, J.-C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
17. Grossman, S., et al.: Online detection of effectively callback free objects with applications to smart contracts. In: Proceedings of the ACM on Programming Languages 2(POPL), pp. 1–28 (2017)

18. Hildenbrandt, E., et al.: Kevm: a complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022
19. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography Workshops (2017)
20. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: NDSS, pp. 1–12 (2018)
21. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
22. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269. ACM (2016)
23. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf
24. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653–663 (2018)
25. O'Connor, R.: Simplicity: a new language for blockchains. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, pp. 107–120 (2017)
26. Pass, R., Shelat, A.: Micropayments for decentralized currencies. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015: 22nd Conference on Computer and Communications Security, pp. 207–218. ACM Press, Denver (2015). https://doi.org/10.1145/2810103.2813713
27. Pass, R., Shelat, A.: Micropayments for decentralized currencies. Cryptology ePrint Archive, Report 2016/332 (2016). http://eprint.iacr.org/2016/332
28. Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. Master's thesis. Chalmers University of Technology, Sweden (2016)
29. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997)
30. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.: Security: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, pp. 67–82. ACM, New York (2018). https://doi.org/10.1145/3243734.3243780
31. Wohrer, M., Zdun, U.: Smart contracts: security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 2–8. IEEE (2018)
32. Zhou, E., et al.: Security assurance for smart contract. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5. IEEE (2018)