Formal Verification of Smart Contracts

Vittorio Torri, Daniele Paletti

April 2020

1 Introduction

On April 2016 *slock.it* starts a venture capital DAO [8], called *The DAO*. Two months later an attacker exploits a recursive calling vulnerability [7] and steals $55 million from *The DAO* in front of anyone who cares and cannot be stopped.

Security and verification in Smart Contracts from that moment on becomes a central topic. Enforcing properties over computer programs is an hard task with all the limitations coming from important results in theoretical computer science. Smart contract formal verification is concerned with verifying specific properties of smart contracts in order to reduce risks when deploying on the blockchain, which due to its architecture does not allow deploying updates.

Throughout this paper the state of the art in smart contracts formal verification will be examined. This field is still very young and many approaches are currently being explored through a variety of tools. Mapping different attempts and their peculiarities may help in building bridges among different efforts and in identifying the most promising research directions.

2 Background

2.1 Blockchains

A blockchain is a P2P network that timestamps each transaction, hashes it on a chain of hash-based proofs and then forms a record that cannot be changed without redoing the proof [25]. Blockchains rely on a distributed ledger, that is a consensus of replicated, shared, and synchronized digital data spread across multiple entities. Blockchains were originially devised to avoid double spending without relying on third parties, financial institutions in particular. Many blockchain implementations exist, the one we will take into consideration is the Ethereum Blockchain because it is the most prominent platform for smart contract development. Ethereum is an open-source, public, blockchain-based distributed ledger featuring smart contract (scripting) functionality. Ether (ETH) is the native cryptocurrency used on the Ethereum network and is used to compensate miners who secure transactions. Ether is required to transact on the Ethereum network.

2.2 Smart Contracts

A smart contract is a computerized transaction protocol that executes the terms of a contract [34]. On the blockchain a smart contract can be seen as a piece of code that runs on the blockchain and is guaranteed to produce the same result for everyone who runs it. The Ethereum blockchain

allows Smart Contract specification through Solidity. Solidity is a Turing-complete, object-oriented, high level language which targets the Ethereum Virtual Machine (EVM). The EVM is the runtime environment for Ethereum smart contracts and it is deployed on each Ethereum node, follows that any operation on the EVM consumes resources on the hosting machine. In order to prevent "overload" of the hosts, each operation on the EVM consumes a certain amount of gas, the unit for how much computation work is done. The price of gas is regulated through a fee market in which every user decides how much is willing to pay for each unit of gas, follows that the cost of each transaction on the ethereum blockchain can be calculated as $GasUsed * GasPrice$.

2.3 Petri Net

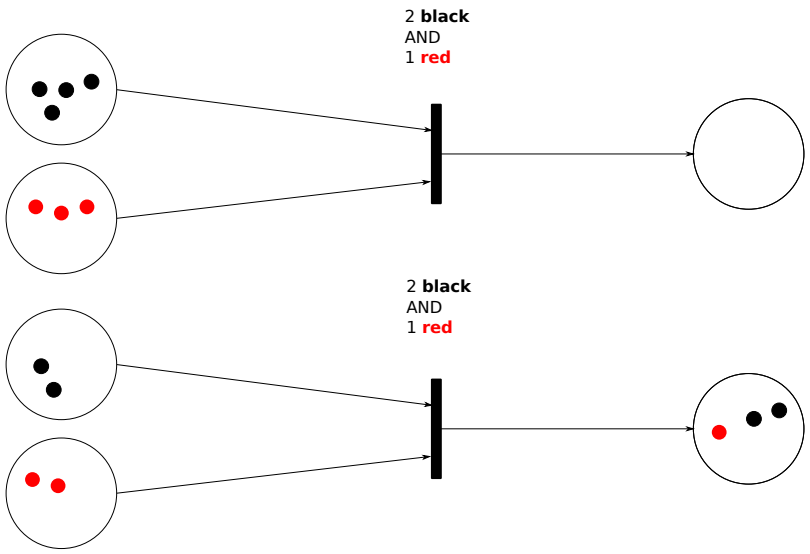


Figure 1: Example Coloured Petri net

A Petri Network is a directed bi-partite graph where:

- nodes represent either transitions or places
- directed arcs represent which places are pre/post-conditions for which transitions

In general places may contain a certain number of tokens, the peculiarity of Coloured Petri Nets is that those tokens may have a data type attached. Usually places contain markings of the same type but this is not an actual constraint of the model. Transitions are said to 'fire' if a condition over the tokens of their input places is satisfied as in Figure 1. Firing conditions may employ various specification languages.

Colours are only a conceptual framework to reason about types, any type is actually allowed such as Integers, String and Custom defined compound types. Coloured Petri Nets (CPNs) models can get arbitrarily complex, in [13] many more details are given about the tool and some distributed systems applications are outlined.

2.4 Behavior-Interaction-Priority Framework

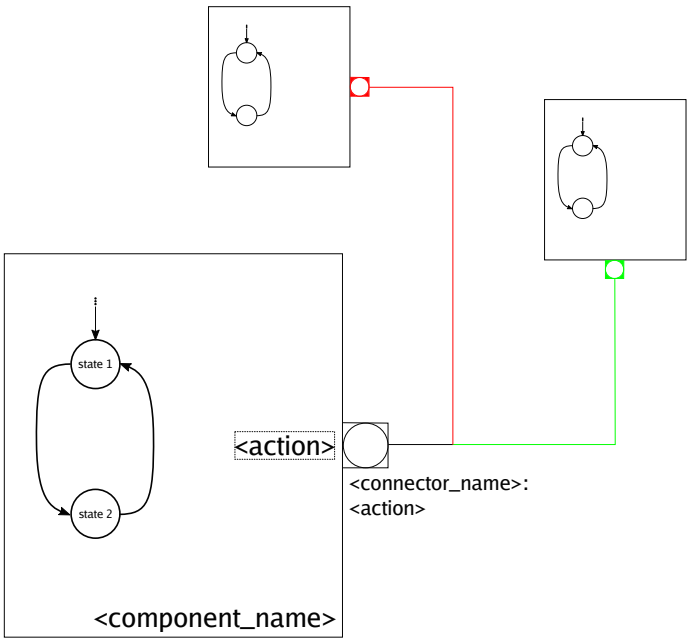


Figure 2: Generic BIP model

Behavior-Interaction-Priority (BIP) is a general framework encompassing rigorous component-based design. A component in the BIP framework is an extended finite state automata (FSA) with:

- variables: used to store local data;
- ports: used to model interaction among components, associated with an action that is a computation over local variables;
- connectors: used to represent sets of interaction patterns.

FSA transitions are also extended with:

- boolean conditions: need to be satisfied to move from a state to another, no condition means branch always taken;
- actions: operations carried out while traversing a transition;

A general representation of a BIP architecture can be seen in figure 2 where multiple components are connected through ports and connectors. Actions carried out on connectors usually are computations over local variables of the two involved components so as to map the value of one to the other, sort of a message passing interface. Actions on ports, inside the dashed rectangles, usually are computations that assign a certain value to the local variables.

2.5 Markov Decision Process

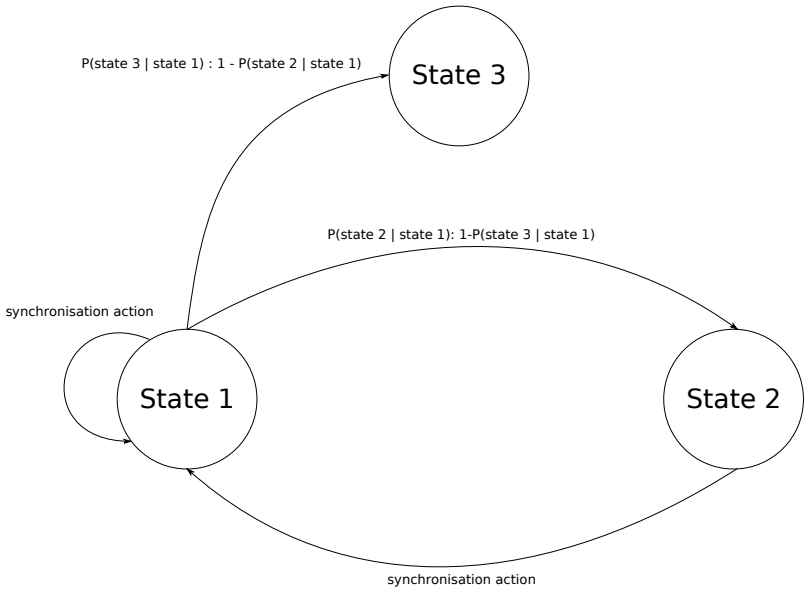


Figure 3: Example of a Markov Decision Process

A Markov Decision Process (MDP) is an extension over FSAs with transitions that can be traversed with a given probability as shown in Figure 3 . More precisely over a directed edge we find a tuple "a:b" where "a" is the probability of traversing the current edge and "b" is the probability of traversing the other arch outgoing from the current state (more than two possible options may arise in decisions, to encode so it is sufficient to build longer tuples). In figure 3 there are also synchronisation actions which are to be performed together between two or more automata. MDPs in fact model multiple decision takers through multiple FSAs and bridge them together through synchronisation actions.

2.6 Interface Automata

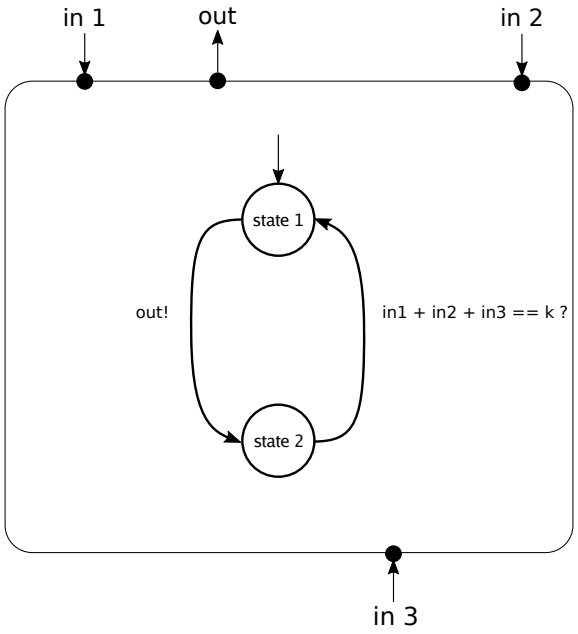


Figure 4: Example of Interface Automata

Interface Automata (IA) are introduced in [5]. The model is based on FSAs extended with input/output interfaces as in figure 4 . An output in a IA represents a method call, that means that an output in one IA must always have a corresponding input in another IA which represents its implementation. Inputs instead can be used to drive the internal logic of the IA.
As we can see in figure 4 the three inputs are used to check whether their sum is 3 (pretending they have a type for which this makes sense) using a notation such as that the expression to be checked is followed by question mark.

3 Related Work

This is not the first survey on this topic. In [24] Murray and Anisi present some approaches, others are presented in [10] by Harz and Knottenbelt, while in [19] J. Liu and Z. Liu considered a large number of papers on the topic, but they didn't investigate too much the single proposals. The main contribution of this paper is the empirical test of some of the approaches, presented in section 6 where we test them on two real smart contracts, gaining a better understanding of their expressiveness and usability. Moreover the role of the blockchain architecture in the verification process is explored (see section 7).

4 Review Methodology

We have identified the reviewed approaches to smart contract verification by looking for papers related to the keywords *"Smart contracts formal verification"* and following the *Related work* sections of these papers. This survey does not pretend to encompass all existing approaches to the problem, but for sure a large part of them, the most promising ones, taking also into consideration the heterogeneity in the approaches followed.

We will present all the approaches placing a particular emphasis into their level of practical usability, the availability of the related tools and the coverage of Solidity syntax. A final summary table is presented in section 5.13.

Eventually some of the approaches are tested on two reference smart contracts in section 6.

5 Review Result

5.1 Promela

In [27] the authors present a model-checking approach for the verification of Solidity smart contracts based on SPIN and its language Promela[12]. SPIN is a well-known model checker, born in 1989, based on the construction of non deterministic finite state automata (NFA), on which it is able to verify assertions, LTL properties and liveness properties. Among the interesting features of SPIN there are various options to improve the ability to deal with large state spaces, in both exhaustive and approximate ways.

In this proposal the authors start defining a grammar for a large subset of Solidity and a grammar for the useful subset of Promela, then they define the translation between the two grammars. The idea is to be able to perform this translation automatically and with this purpose they extend the Solidity syntax with particular comments which will be translated in the Promela assertions which need to be verified. They have implemented a Python prototype of the translation tool, but it does not seem to be available. They present an evaluation example with a very simple coin contract, on which they are able to verify some basic assertions, showing also the case when they fail due to errors in the code.

The basic idea of the mapping between Solidity and Promela is to translate each smart contract method into a Promela process. In the Promela `init` section a certain number of processes are spawned and they will cover all possible interleaved executions of the contract methods. The body of each method is executed atomically, since in the blockchain two distinct transactions on the same smart contract are never executed in parallel. The parameters of the methods are initialized in all possible ways. The approach is focused on the expression of assertions which must hold, to guarantee the correctness of the Solidity code. We have tested this approach with other smart contracts in section 6.2, where it is analyzed more in deep.

5.2 Why3

In [26] an approach based on the *Why3* tool is proposed. Why3 is a deductive verification tool, it provides a language called *WhyML* and it relies on external theorem provers. The authors use an energy market Solidity smart contract as example and they develop their prototype on this, avoiding to consider all the Solidity syntax.

In their encoding of smart contracts they distinguish between public and private methods, where the first ones are those which can be called by the external oracles. For private functions they define pre and post conditions, while for public functions they cannot control the input, so they define exception to be raised at runtime in case of illegal input data. Their idea is to write directly the contracts in WhyML at the high level and then compile them into EVM. For this purpose they developed a compiler from Why3 to EVM.

The main weakness of this proposal is related to the absence of a systematic definition of the translation from Solidity, or from the considered subset of Solidity. This is probably due to the idea of using WhyML as a substitute for Solidity, but it is not clear how this should be done without having a precise correspondence between the two languages. In addition, even if they start from this idea, this approach could be useful also for the verification of already existing Solidity smart contracts, and in effect they started from a Solidity smart contract for their example, but the absence of its Solidity code, except for a couple of methods, makes the correspondence less clear. They also adopt the hypothesis of being in a private blockchain and this eliminates the need to consider some aspects of Solidity, such as fallback functions. This is another relevant limitation, even if this is not the only approach which does not consider some aspects more related to the blockchain architecture.

The positive aspect of this approach is that it allows to express various properties, even those related to the state evolution of smart contracts' fields, in an easier way than other approaches. For instance here is shown how to guarantee that the sum of two fields remains unchanged:

```
1 ensures {(old marketBalanceOf[market]) + (old importBalanceOf[_to]) =
    marketBalanceOf[market] + importBalanceOf[_to]}
```

Another interesting aspect of this proposal is the consideration of gas consumption. It is not very clear how this can be used, but anyway is an element which has not been considered in many approaches. In synthesis it seems to be a good approach, but it is still in a initial stage, it can be improved for sure. The idea of directly writing the contract in WhyML does not seem to be so attractive, it should probably be more successful to develop an automatic translator from Solidity to WhyML.

5.3 Scilla

In [31] the authors present *Scilla*, an intermediate language for smart contracts verification. *Intermediate* means that Scilla is not aimed to be an high level language like Solidity but at the same time it requires to be translated in a lower level language to perform the verification. Scilla models smart contracts as *communicating automata*: each contract has a state, partially mutable, and a set of transitions. The state corresponds the contract's fields in Solidity, while each transition corresponds to a Solidity method. Each transition can change the state of the contract and can send a message at its end. A transition is activated by an incoming message sent to the contract, with a tag specifying the transition, and cannot call another transition before the end, when it can send a message which calls another transition. To maintain the possibility to call another smart contract in the half of a function, as it is commonly done in Solidity, the final message sent by a transition can specify a *continuation*, which is a sort of particular transition which receives the return value of the external called method and then sends another message, being able in this way to resume the execution. The authors plan to develop an automatic translation from Solidity to Scilla, but actually it needs to be done manually. To allow formal verification Scilla relies on the Coq verification tool. A manual translation from Scilla to Coq's language Gallina is presented for

the example contract of the paper and examples of liveness (always valid) and temporal (related to different states) properties are provided with it. One example is the following, which states that a crowdfounding contract cannot loose founds until the campaign is terminated:

```
Definition balance_backed st : Prop :=
(* If the campaign has not been funded... *)
¬  funded (state st) →
(* the contract has enough funds to reimburse all. *)
sumn (map snd (backers (state st))) <= balance st.

Theorem sufficient_funds_safe : safe balance_backed.
```

Property definitions rely on some basic common definitions, such as `safe` for safety properties like this, which states that if the property holds in a state before a transition it must hold also in the state which is reached after the transition.

In the paper they use a crowfounding contract, for which they provide the Scilla and the corresponding Coq code, but not the Solidity code. Actually the manual translation from Scilla to Coq does not support *continuation* and Scilla itself does not support loops and exceptions. These are very important limitations, even if the authors plan to overcome them in the future. For sure at the moment this approach is not mature enough, even if it could be improved.

5.4 F*

In [3] Bhargavan *et al.* present two tools for the formal verification of smart contracts based on the F* verification language: Solidity* and EVM*. They translate respectively Solidity high-level code and EVM low-level code into F*. The portion of Solidity syntax supported by Solidity* is very basic, it includes recursion but it doesn't include loops. They provide an example of property specification for Solidity* which is simply a pattern checking, in particular they verify if the return of the send function is checked (it can fail without throwing an exception). EVM* introduces in the translation a burn function to represent the consumption of gas and in the paper an example shows the specification of an upper bound on the gas consumption of a method. The authors have develop a prototype of the two tools, but they are not publicly available. Considering the very limited supported syntax, these tools are still too limited to be of practical use: they have been able to translate only 46 of 396 contracts and in larger contracts datasets the percentage of contracts with loops arrives up to 93% [14].

5.5 K

In [30] and [28] Rosu *et al.* propose the adoption of the *K framework* [33] for the verification of smart contracts. K is a framework which allows to define the formal syntax and semantics of a language and given this it is able to generate various tools (parser, compiler, test-case generator, model-checker,...) which can help in the verification and analysis of software written in that language. The authors, in collaboration with the company *IOHK*, have written a formal specification of EVM, called *KEVM* [11]. They also have defined the formal semantics of a new smart contract programming languages provided by the Ethereum foundation, *Vyper*, which still compiles to the EVM. Finally they have defined a new VM for smart contracts, called *IELE*, whose code has been automatically generated by K, given a formal specification. IELE supports Solidity smart contracts and they are working to extend the support for *Plutus*, a new high-level language for smart contracts.

About formal verification the focus is on the K's reachability logic theorem prover, which combines symbolic execution and reasoning. Reachability logic is the foundation of K and properties to be verified are expressed in term of reachability rules. A reachability rule is a rule of the type $\alpha \implies \beta$, where α and β are static logic formulas, in particular for K they are formulas of a subset of matching logic. In practice α and β represents sets of program status and a rule of this type impose that starting from a state in α the program will arrive in a state in β or it will not terminate.

A relevant aspect is the fact that KEVM is based on EVM, not on Solidity. This implies that the specification of properties is based on the compiled EVM code and this is one of the causes of the complexity of this specification.

For instance, the specification of the `approve` method of the *HackerGold* ERC-20 contract (*HKG*) is the following[1]:

```
[approve]
statusCode: _ => EVMC_SUCCESS
output: _ => #buf(32, 1)
callData: #abiCallData("approve", #address(SPENDER), #uint256(VALUE))
log: _:List ( .List => ListItem(#abiEventLog(ACCT_ID, "Approval", #indexed(#address(
    CALLER_ID)), #indexed(#address(SPENDER)), #uint256(VALUE))) )
refund: _ => _
storage: M1 => M2
origStorage: M1
requires:
    andBool 0 <=Int SPENDER    andBool SPENDER    <Int (2 ^Int 160)
    andBool 0 <=Int VALUE      andBool VALUE      <Int (2 ^Int 256)
ensures:
    ensures select(M2, #hashedLocation({COMPILER}, {_ALLOWANCES}, CALLER_ID SPENDER)
        ) ==Int VALUE
    andBool M1 ==IMap M2 except
        (SetItem(#hashedLocation({COMPILER}, {_ALLOWANCES}, CALLER_ID SPENDER)) .Set
            )
```

Among the interesting features of this approach we can see the verification of *events* and also the possibility to indicate bounds on gas consumption (not present in the above example). More in general it is probably the approach which is more adherent to the EVM language, the coverage of the syntax is complete and there are no risk of approximations or missing parts such as in other models, but on the other side this makes it quiet complex to be used, even the installation of the suite has given us some problems. If it will be improved on this side, and this is in the authors' purposes, it could become more used.

5.6 Securify

In [35] the authors present *Securify*, a software tool for the verification of security properties of smart contracts. The tool takes as input the Solidity or EVM code of a smart contract and it extracts from it a *Stratified Datalog* representation of its Control Flow Graph. Given this and some security patterns to be verified, which can be indicated combining some predefined datalog facts, the tool exploits a Datalog solver to verify the presence of this patterns.

There are two types of patterns: compliance and violation patterns, which respectively verify the absence or the presence of the security problem. Checking these properties precisely is impos-

[1]The complete specification can be found at `https://github.com/runtimeverification/verified-smart-contracts/blob/master/erc20/hkg/hkg-erc20-spec.ini`

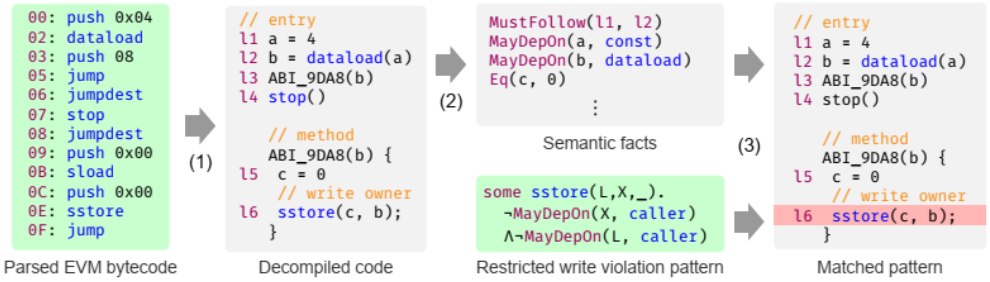


Figure 5: Example of Securify verification process from [35]

sible being smart contracts languages *Turing Complete* and because of this the defined patterns over-approximate the security properties. The authors explain what are the typical security vulnerabilities which can be encountered in a smart contract and that can be verified by Securify. The main ones are:

- *Ether liquidity*: liquidity can be frozen in a contract if it relies on external library to withdraw it (see [9])
- *Reentrancy bug*: this is the bug which caused the famous DAO attack of 2016; it was based on the call to the contract caller function before updating the caller balance in the contract, allowing the caller to withdraw an excessive amount of money.
- *Restricted write*: some fields of a smart contract should be written only by a certain user (*eg*: only the actual owner can change the ownership), the absence of this control could be very dangerous.

Limitations are related to the assumption that all instructions in the code are always reachable, which is not true in general, and to the approximations made in the specification of the patterns, which cause the possibility that a property is not verified as neither hold nor as not hold, leaving a warning for a manual check by the user.

An example of datalog violation pattern for the *No writes after call* property is the following:

```
1 some call(L1, _, _, _). some sstore(L2, _, _). MustFollow(L1, L2)
```

The fact `MustFollow(L1,L2)` is one of the basic datalog semantic facts used by Securify. The first step of the tool is to extract from the contract source code all the semantic facts and then it verifies the indicated properties, such as the one above, against them. In this case the approximation is due to the fact that each write that follows as a call is considered a violation, but it's not always true. The problem typical arises when there is a call whose transferred value depends upon a field which is modified only after the call.

The Securify chain process is shown with another example in figure 5.

They compared Securify with two other tools, *Oyente*[21] and *Mhitryl*, based on symbolic execution and able to verify some of the properties verified by Securify. The performance of Securify are better, in particular it doesn't have the high number of false positive that these tools have and

it never completely miss a vulnerability, in the worst case it marks it as a warning to be manually verified. For this reason we've not considered in detail these two other tools in our analysis.

An online version of the tool is available[2].

5.7 Zeus

Another tool for the verification of security properties of smart contracts is *Zeus* [14]. The core of Zeus is the translation of the smart contract high level code into *LLVM*, a lower level language for which formal verification tools already exist. After the translation Zeus uses *Seahorn* as tool for the verification, but other tools can substitute it, such as *SMACK*. The *policies* to be verified are expressed as 5-tuples `<subject, object, operation, condition, result>`, where the subject indicates the variables to be considered, the `object` are the other variables involved in the interaction, the `operation` is the function on which the policies are verified and the `result` indicates if the condition must be respected or avoided. This tuples are then automatically translated into assertions, which are placed into the code by looking at subject, object and operation. In the insertions of assertions a conservative approach is chosen and it's here that false positives may arise. The tool supports most of the Solidity syntax but it has some limitations in the handling of gas consumption, which is not present in LLVM. It also does not support the use of assembly in Solidity (rarely used in smart contracts) and the `super` keyword. The tool is mainly focused on security properties, similarly to Securify: it allows to check for properties like reentrancy, transaction order dependence, overflows/underflows, which are managed by the insertion of specific global variables and assertions, but anyway the specification of the properties previously explained make it quiet flexible. A test on a big number of contracts about these security properties produced 0 false negatives and only a small number of false positives. Unluckily this tool has not been released, it would have been interesting to compare it with Securify.

5.8 EthRacer

A particular tool which is focused on a specific class of smart contracts problems is *EthRacer*[15]. This tool is focused on the so called *Event Ordering (EO) bugs*, i.e bugs related to the change in the transaction execution order. The atomicity of transaction execution in Etheurem, i.e. the absence of explicit concurrency, does not provide any guarantee on the order in which transactions are processed by miners and this is a not rare source of bugs. In this work the invocations of contract functions are defined as *events* and their consequent executions are referred as *traces*. The tool exploit a *Dynamic Symbolic Executor (DSE)* to enumerate all possible events and explore them, but this clearly leads to a combinatorial explosion in the number of paths to be traversed. To avoid this problem the tool performs a series of optimizations. The most important is the identification of the so called *happens-before (hb)* relationship between two events: if two events can occur only in one order because the opposite leads to an exception, only the paths which respect this order needs to be explored. In practice they identify a weaker version of the relationship, which involves only two consecutive events. They also avoid to consider permutation of events which do not read and write the global state of the contract. In addition permutations of events in which the change is only between equal functions but with different inputs are checked only at the end, since they are often sources of apparent bugs which correspondent in practice to a desired behaviour.

[2] `https://securify.chainsecurity.com/`

A particular attention is devoted to contracts which make use of the *Oracolize API* to call asynchronously external oracles: for them the *linearizable traces*, i.e. traces in which the natural execution order of transactions is respected, are taken as the correct behaviour: if other traces leads to a different behaviour they are considered as a bug. EthRacer needs in input the smart contracts EVM or Solidity code but also a fully synched Ethereum blockchain where the contract is deployed to get its initial state.

The tool enumerates all traces up to a given bounded length and then proceeds to the analysis with the previously explained optimizations. It is able to consider 90% of EVM op codes.

An interesting feature is the fact that the tool returns a proof the EO bugs found, that can be easily manually analyzed. In their test about 50% of discovered EO bugs could be considered intentional behaviours. In general is not easy to define when a different behaviour on different execution orders can be considered intentional or still correct and when it is not desirable.

Compared to *Oyente* it has been able to find a double number of bugs and all the ones reported by Oyente. A current limitation is that only the changes to the local contract state are considered to identify different behaviours, but in reality there also situations in which only the state of other contracts or the logged events are altered.

Other tools such as Securify can only intercept a small subset of this kind of bugs. In particular Securify consider only the following cases:

- the amount of a transfer depends on a writable field of the contract,
- the recipient of a transfer depends on a writable field of the contract,
- the execution of a transfer depends on a writable field of the contract,

Even if it is not able to find general EO bugs, the second case falls in the ones which are not caught by EthRacer, so even in this category of bugs the two tools are actually complementary.

5.9 Coloured Petri Networks

A Petri Network is a mathematical modelling language conceived for distributed systems modelling, its adoption in smart contract verification relies mainly on the implicit non determinism that characterises the model. Coloured Petri nets are specifically chosen because they are well versed when it comes to modelling:

- communication,
- synchronisation,
- resource sharing.

Moreover Coloured Petri Nets while still being backward compatible with standard Petri Nets are much more flexible when it comes to modelling *articulated* systems which can be formally checked with well established verification and simulation tools [13, 36] . In [20] Liu Z. and Liu J. demonstrated how a multi layered CPN may be used for not only modelling the smart contract itself but also accounting for user interaction and attacker models. In a layered CPN substitution transitions are introduced, those transitions represent sub-nets in order to give a multi-layered representation of the whole system. The Top Layer of the CPN represents the interface of the contract.

The case examined in [20] was that of a bank account which allowed withdrawing and depositing, represented in the top level as substitution transitions. Specifying the substitution transitions accounts for the specification of a certain functionality.
CPNs allow modelling the attacker seamlessly as an integrated component of the system. First of all the attack behaviour needs to be established, in [20] the attacker would modify the fallback function of the contract (a function that the EVM introduces to be called when an operation is carried out without any function call) and start withdrawing money from the account in a malicious manner. Adding an attack substitution transition inside the attacked functionality is enough to model the attack, at that point the attack sub-net encoding the attacker behaviour it is to be defined.
CPNs can be simulated step by step (even manually in simple cases) or one of the already existing tools can be used. In [20] State Space tools are used to examine model correctness and then simulation tools (ASK-CTL and CPN-ML) are used for mimicking attack interactions and system evolution under attack. As we can clearly see from [20] even with toy contracts the network tends to become complex and reasoning about it can get clumsy. Smart contract specification language to CPN translation is a hard task when applying this method to non trivial smart contracts. As earlier stated CPNs tend to become very large for very simple contracts, one approach that could be investigated designing the smart contract (or critical portions) directly as a CPN and then automatically generate code [23, 16] from it. While a limiting approach code generation may give a structured and verifiable development process to smart contracts.

5.10 Behavior-Interaction-Priority Framework

The Behaviour Interaction Properties (BIP) framework is a component based modelling approach for statistical model checking. Component based approaches are particularly advisable in modelling smart contracts as many moving parts are involved and interaction modelling is favoured. The blockchain together with the smart contract we are modelling, users and attackers can be seen as a stochastic system and as such statistical model checking is an advisable method. Statistical model checking can be either carried out through approximate solutions of ad-hoc logical models or through hypotesis testing. BIP allows hypothesis testing which is the approach that less relies on the class of systems being analysed [18] thus being more flexible. Smart contracts can be modelled in BIP either as single components or as a multi-component system depending on their complexity, BIP application to smart contract verification is explored in [1] . Remarkable in the approach outlined in [1] is the way the blockchain and the attacker can be effortlessly introduced in the model without affecting the way the example smart contract was originally modelled. A schematic representation of how the blockchain was modelled in [1] is showed in figure 6 where several simplifications were adopted. The model in 6 given its modularity can be extended to also take into account smart contracts interactions, multiple miners and gas (for ethereum blockchain). Some details are to be noted in order to understand the relevance of the model:

- the two data ports Pending TX and Blocks are sufficient to represent the export of transaction lists and blocks;
- Get TX is the entry point for user interaction which comes from a separate user block that models user behaviour;
- an attacker is modelled through another separate component which encodes its behaviour and all its interactions with the user and the blockchain are represented by connectors.

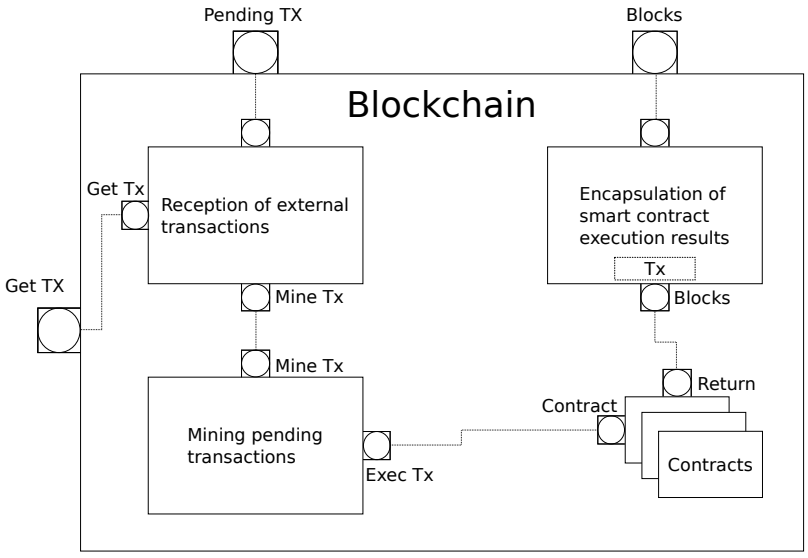


Figure 6: Schematic representation of the blockchain model in [1]

Once the model is synthesised, statistical model checking is applied over attack scenarios so as to calculate the probability of success of an attack and derive tweaks to the smart contract. While the modelling formalism is very powerful and extensible the model checking phase is not as powerful. The approach used in [1] is that of manually writing attack scenarios and evaluate their probability, this is not that easy outside of toy examples. Moreover FSA may not be sufficiently powerful to model behaviours that are inherently stochastic. The BIP framework may benefit from diversity among its core components and so adding a component able to encode probabilistic process, swapping FSA and Markov Decision Processes for example, may give more descriptive power to the framework. Extensions may render the model checking process more complicated for example adding a probabilistic component may need mixing probabilistic and statistical model checking.

5.11 Markov Decision Process

A Markov Decision Process (MDP) is a description of a decision-making process that may depend on non-deterministic random choices. A MDP given its definition is clearly a natural choice for modelling user behaviour and user to smart contract interaction. On top of that MDPs allow probabilistic model checking which is a well versed technique for stochastic systems [17]. In [4] Bigi et al. use MDPs together with Game theoretical modelling to model and verify a toy smart contract encoding a simple goods market. Non-cooperative strategic games are used in a preliminary phase to gain as much information as possible to mark MDPs edges with sensible probabilities. Main outcome of the game theoretical section are the Nash equilibrium profiles of the game through which assumptions can be made on the probability of different choices taken by the parties. All the actors in the smart contract are modelled through their own MDP and probabilities are tailored to the aforementioned game theoretical observations. Probabilities in [4] are parametrised in order to easily simulate different tendencies in the behaviours of the actors. The model is validated through the probabilistic model checker PRISM that derives the probabilities of statements. Statements to

be analysed through PRISM must be manually crafted.

The approach in [4] is not very clear in stating how one would generalise over non-toy contracts the role that game theory played in their example. The model used was very basic yet created a relatively large extended form game. Moreover the benefit that game theoretical modelling brought may have been well gained through empirical study of application usage or, in case the smart contract was not yet deployed, one could have directly guessed "correct" probabilities in MDP through repeated runs of PRISM on ad-hoc statements. Probabilistic model checking suffers then of the same weaknesses highlighted for Statistical Model Checking being that statements, in other words scenarios, need to be manually fed to PRISMA and determining attack scenarios may not be immediate for complex smart contracts.

The approach outlined in [4] connects many fields but builds very unstable bridges. A game theoretical analysis may be well beneficial to the whole formal process but probably one should resort to algorithmic game theory techniques to build semi-automated mechanisms so as to outline a more structured analysis process. On top of that the result of the preliminary analysis should guide the subsequent analysis more closely maybe constraining the MDP structure. Automatic statement generation for PRISMA is also a very promising approach to improving the whole process.

5.12 Interface Automata

An Interface Automata (IA) is a FSA extended with input and output ports thus allowing to build a network of IA well versed for component based design and analysis. As per the BIP framework component based approaches really shine when modelling smart contracts in their environment coupled with user and attacker models. Moreover IA are a light weight but strict formalism that captures temporal aspects of software interfaces [5] thus very well suited to model smart contract execution over time together with the subsequent user decisions and blockchain evolution. In [22] IAs are used to model smart contracts and user behaviour. The smart contract analysed in [22] conceived two agents one issuer and one consumer both of which are modelled through a specific IA encoding their behaviour. Another IA is used for modelling the transaction, that is the smart contract logic. The blockchain in this case was not modelled but the strong component based approach allowed modelling all other components seamlessly and just leaving some IA outputs unmatched.

This model, even if no implementation of the blockchain was presented, showed idempotency with regard to the BIP framework. Model checking is carried out with NuSMV wich is a symbolic model checker based on binary decision diagrams (BDD), which sets this method apart from the aforementioned BIP framework which instead uses statistical model checking.

One promising approach is taking advantage of the IA model by using verification tools specifically built for it [6] in order to skip the translation process and employ ad-hoc algorithm that may foster more investigation in using IA as a software engineering tool for developing smart contracts.

5.13 Final Comparison

In Table 1 a final comparison of main characteristics of all the approaches is reported.

Approach	**Type**	**Syntax coverage**	**Availability**	**Ease of use**	**Notes**
Promela	Model-checking	High	Manual translation	Medium	Some blockchain peculiarities are lost in the model
Why3	Theorem prover	Medium	Manual translation	Medium	Compiler from WhyML to EVM, but not completely defined translation between Solidity and WhyML
Scilla	Theorem Prover	Medium-Low	Double manual translation	Low	Actually present many limitations which may be overcome in the future
F*	Theorem prover	Low	Not available	Medium	Actually very limited, it could be improved. Supports both Solidity and EVM input
K	Symbolic execution and model-checking	Complete	Available, requires to write specifications in K language	Low	Promising, but it seems hard to use. EVM as input
Securify	Bug verification	Complete	Available	High	Useful, but with some false positives
ZEUS	Bug verification and symbolic execution	High	Not Available	Medium	Seems good, but it is not available and cannot be compared against Securify. Can be extended to other high level languages for smart contracts
EthRacer	Bug verification	High	Available	Medium	Focused on a particular class of subtle bugs, useful but requires to manually analyze results
CPN	Model-Checking	Complete	Manual Translation	Low	Methodology demonstrated on a very simple Smart Contract
BIP	Model-Checking	Complete	Manual Translation	Low	A simple but extensible blockchain model is provided
MDP	Model-Checking	Complete	Manual Translation	Low	Writing transition probabilities may be challenging in real-world scenarios
IA	Model-Checking	Complete	Manual Translation	Low	IA are to be translated in NuSMV manually

Table 1: Summary comparison of the various approaches for smart contracts verification

6 Empirical Evaluation

6.1 Reference smart contracts

In order to test some of the previous approaches we chose two smart contracts. We selected them with the idea to use real world smart contracts, not excessively complicated, but more than the typical toy-examples used in the reviewed papers.

The first one is *Pyramid* [29], a quiet simple pyramidal scheme, which is representative of a wide number of similar contracts present in the blockchain [2]. A new user joins the scheme investing 0.1-5 ETH which are converted into queue entries, one queue entry every 0.1 ETH. Every time the queue reaches an odd length, the second to last user receives 194 finney. It works well as long as new users continue to join it. Regarding the `join` method one could test the correct payment of previous users. On top of that the contract balance needs also to be checked as it must be ≥ 200 in order to allow payment of old users on new joins, in this regard the `collectFee` method may rise some concerns.

```
pragma solidity ^0.4.0;

contract Pyramid {
    address master;

    address[] memberQueue;
    uint queueFront;

    event Joined(address indexed _member, uint _entries, uint _paybackStartNum);

    modifier onlymaster { if (msg.sender == master) _; }

    function Pyramid() {
        master = msg.sender;
        memberQueue.push(master);
        queueFront = 0;
    }

    // fallback function, wont work to call join here bc we will run out of gas
        (2300 gas for send())
    function(){}

    // users are allowed to join with .1 - 5 ethereum
    function join() payable {
        require(msg.value >= 100 finney);

        uint entries = msg.value / 100 finney;
        entries = entries > 50 ? 50 : entries; // cap at 5 ethereum

        for (uint i = 0; i < entries; i++) {
            memberQueue.push(msg.sender);

            if (memberQueue.length % 2 == 1) {
                queueFront += 1;
                memberQueue[queueFront-1].transfer(194 finney);
            }
        }

        Joined(msg.sender, entries, memberQueue.length * 2);

```

```
        // send back any unused ethereum
        uint remainder = msg.value - (entries * 100 finney);
        if (remainder > 1 finney) {
            msg.sender.transfer(remainder);
        }
        //msg.sender.send(msg.value - entries * 100 finney);
    }

    function collectFee() onlymaster {
        master.transfer(this.balance - 200 finney);
    }

    function setMaster(address _master) onlymaster {
        master = _master;
    }

}
```

Listing 1: Pyramid source code

The second one is *WETH9* [37], a contract which allows to convert ETH in wETH tokens compliant with the ERC-20 standard. This smart contract is among the ones with highest balances in Ethereum, actually the third one in this ranking.[3]. The contract has two fields, `balanceOf` and `allowance`. The former keeps track of the balance of each user, while the latter records which users want to allow another user to spend some of her tokens. The methods `withdraw` and `deposit` operate on `balanceOf`, while `approval` operates on `allowance`. The methods `transfer` and `transferFrom` allow exchanging tokens among the users.

```
pragma solidity ^0.4.18;

contract WETH9 {
    string public name     = "Wrapped Ether";
    string public symbol   = "WETH";
    uint8  public decimals = 18;

    event  Approval(address indexed src, address indexed guy, uint wad);
    event  Transfer(address indexed src, address indexed dst, uint wad);
    event  Deposit(address indexed dst, uint wad);
    event  Withdrawal(address indexed src, uint wad);

    mapping (address => uint)                       public  balanceOf;
    mapping (address => mapping (address => uint))  public  allowance;

    function() public payable {
        deposit();
    }
    function deposit() public payable {
        balanceOf[msg.sender] += msg.value;
        Deposit(msg.sender, msg.value);
    }
    function withdraw(uint wad) public {
        require(balanceOf[msg.sender] >= wad);
        balanceOf[msg.sender] -= wad;
        msg.sender.transfer(wad);
        Withdrawal(msg.sender, wad);
```

[3] `https://etherscan.io/accounts` - URL consulted on 18th April 2020

```
28      }
29
30      function totalSupply() public view returns (uint) {
31          return this.balance;
32      }
33
34      function approve(address guy, uint wad) public returns (bool) {
35          allowance[msg.sender][guy] = wad;
36          Approval(msg.sender, guy, wad);
37          return true;
38      }
39
40      function transfer(address dst, uint wad) public returns (bool) {
41          return transferFrom(msg.sender, dst, wad);
42      }
43
44      function transferFrom(address src, address dst, uint wad)
45          public
46          returns (bool)
47      {
48          require(balanceOf[src] >= wad);
49
50          if (src != msg.sender && allowance[src][msg.sender] != uint(-1)) {
51              require(allowance[src][msg.sender] >= wad);
52              allowance[src][msg.sender] -= wad;
53          }
54
55          balanceOf[src] -= wad;
56          balanceOf[dst] += wad;
57
58          Transfer(src, dst, wad);
59
60          return true;
61      }
62  }
```

Listing 2: WETH9 source code

6.2 Promela

6.2.1 Model

As already mentioned in section 5.1 this modeling approach is based on a correspondence between Solidity functions and Promela processes. The `init` section of the Promela model contains the contract's constructor code and runs a certain number of processes for the various methods, which will be interleaved in all possible ways. The contract's fields are mapped into Promela global variables, with some limitations in the datatypes: Promela does not support dynamic arrays and mappings, so they need to be translated into static arrays. Also multidimensional arrays are not directly supported, even if there is a workaround that we have applied for them. In particular for the double mapping `allowance` of WETH9 (line 14), we have declared it in the following way:

```
1  typedef array {
2      int value[255];
3  };
4  array allowance[255];
```

The `address` type of Solidity is translated into `byte`, so that we consider 256 different addresses, for simplicity. There are various elements whose translation is not clearly defined in [27] and that are not present in their simple example, but are quiet common in Solidity contracts.

First of all the special variables `msg.sender` and `msg.value`: we have modeled them as they were additional parameters of the contract's methods.

Then the Solidity modifiers, such as `onlyMaster` and `payable`. For the first one we have included the code of the modifier at the begin of the code of the method, and this can probably be applied for all user defined modifiers. `Payable` has been ignored, since we have not used `msg.value` inside non-payable functions, but it could be possible to ensure that `msg.value` is zero if the method is not `payable`, especially in a more systematic translation where `msg.value` could be present in every method, even if not used.

Another very important aspect which has not been covered in the paper is how to deal with Solidity `transfer` function. Here the problem is not anymore related to the contract alone, but it involves somehow the modeling of the users which interact with the contract. We have decided to use a global variable to represent the contract balance, which can be useful also every time that `this.balance` is used, independently from the `transfer` function. Then a global array with 256 records (corresponding to the available addresses in the model) stores the balance of each user (or better, address). Therefor the `transfer` function will be translated with code which moves Ether among these variables. An example of modelled method is provided in listing 3 where the Pyramid `join` method is reported, without the rows related to the verification.

```
1  proctype join(){
2
3      //msg.value
4      #define LOW_join_msg_value 100
5      #define HIGH_join_msg_value 2147483647 //max integer value in promela
6      msg_value = LOW_join_msg_value;
7      do
8           :: msg_value < HIGH_join_msg_value -> msg_value++
9           :: break
10     od
11
12     //msg.sender
13      #define LOW_join_sender 0
14      #define HIGH_join_sender 254
15      byte msg_sender = LOW_join_sender;
16      do
17          :: msg_sender < HIGH_join_sender -> msg_sender++
18          :: break
19      od;
20
21      assert(msg_sender >=0);
22
23     d_step {
24          /*[OMITTED SAVE OF OLD VALUES]*/
25
26          contractBalance = contractBalance + msg_value;
27          balances[msg_sender] = balances[msg_sender] - msg_value;
28
29          join_entries = msg_value / 100;
30          if
31             :: join_entries > 50 -> join_entries = 50;
32             :: else -> skip;
33          fi;
```

```
34          join_i = 0;
35          do
36             :: join_i < join_entries ->
37                 memberQueue[memberQueue_length] = msg_sender;
38                 memberQueue_length++;
39                 if
40                     :: memberQueue_length % 2 == 1 ->
41                         queueFront++;
42                         balances[memberQueue[queueFront-1]] =   balances[memberQueue
                               [queueFront-1]]  + 194;
43                         contractBalance = contractBalance - 194;
44                     :: else -> skip
45                 fi;
46                 join_i++;
47             :: else -> break
48          od;
49          join_remainder = msg_value - (join_entries * 100);
50          if
51             :: join_remainder > 1 ->
52                 contractBalance = contractBalance - join_remainder;
53                 balances[msg_sender] = balances[msg_sender] + join_remainder;
54
55             :: else -> skip
56          fi
57
58          return_label_join:
59    }
60
61 }
```

Listing 3: Model of Pyramid join method in Promela

6.2.2 Verification

Defined these aspects of the translation, we verified some properties. We can take as example the `join` method of the *Pyramid* contract: here we would like to be sure, at least, that no more than 5 ETH have been acquired by the contract balance and that the member queue has been appropriately updated. All these properties require to compare the value of some variables before and after the method execution. In the paper's example there are no properties of this type, but we have been able to express them introducing new *shadow* variables, named with the suffix `_old`, which save, at the begin of the method, a copy of the values of the original variables, so that they can be used at the end in the comparison with the new values that we have after the execution. In listing 4 we reported the code for some of these properties. We omit for brevity the obvious initialization of the shadow variables at the begin of the method's code.

```
1 //memberQueue_length correctly updated
2 if
3    :: msg_value > 5000 -> assert(memberQueue_length == memberQueue_length_old +
         (5000)/100 );
4    :: else -> assert(memberQueue_length == memberQueue_length_old + (msg_value)/100
         );
5 fi;
6
7 //No more than 5000 finney added to the contract balance
8 assert(contractBalance <= contractBalance_old + 5000);
```

```
9
10 //No money disappeared
11 sumOldBalances = 0;
12 sumBalances = 0;
13 i = 0;
14 do
15    :: i > 254 -> break;
16    :: else -> sumOldBalances = sumOldBalances + balances_old[i]; sumBalances =
          sumBalances + balances[i]; i++
17 od;
18
19 assert(contractBalance + sumBalances == contractBalance_old + sumOldBalances);
```

Listing 4: Code for the properties of the join method in the Pyramid contract

Launching the verification in SPIN we encountered some problems related to the size of the state space which requires an high amount of memory to be managed. To overcome this problem SPIN offers a variety of options which can help in managing large state spaces. In particular there are exhaustive search options, such as state compression and *DMA*, and approximate techniques such as *BITSATE* hashing. We first tried the exhaustive options, but they were not able to complete the search. We then tried to simplify the models, reducing the size of arrays, but this has not been enough. We tried the *BITSTATE* technique, which does not guarantee to cover all the states, and in this way we have been able to complete the verification on the Pyramid original model and on the simplified WETH9 model (still not on the original WETH9 model). On the Pyramid model it leads to a state coverage of 97.5%, while on the reduced WETH9 it leads to 100% coverage. These results have been obtained with a desktop pc with 7 GB RAM - Intel i5 2500k.

6.2.3 Evaluation

This type of modelling is able to capture a large part of the Solidity syntax, considering also the constructs for which we have defined here a translation, and it does not require an high effort to create the model starting from Solidity code. An automatic translation tool, such as the one that the authors should have partially developed, could be for sure realized. The model allows also to express a good variety of properties, even if there are some aspects which remains uncovered, such as the interactions with other contracts. However the main problems arise in the verification, where the model complexity quickly become too high to be elaborated with a normal hardware. The *BITSTATE* option of Solidity is for sure a good countermeasure, but can we afford to have a not complete, even if very high, coverage? The answer is probably no. This may be sufficient to state that this approach can't be the solution to the smart contracts formal verification problem.

6.3 Securify

6.3.1 Verification

Giving the *Pyramid* contract in input to the Securify scanner it reported a series of vulnerabilities. We went through them and checked if they were real vulnerabilities:

1. 🟧🟥 *Transaction Order Affects Ether Receiver* at lines 38 and 53: in line 53 it is true that if, for instance, the `collectFee` method is invoked and followed by a `setMaster` call, there is no guarantee that they will be executed in order: it could be the new `master` to receive the fees! Also in line 38 the transfer can be influenced by the order execution of `join`s, but we

can say this is less relevant, considered the purpose of the contract to continuously increase the pyramid and the fact that all users are repaid of the same amount of Ether.

2. 🟧🟥 *Transaction Order Affects Ether Amount* at lines 47 and 53: for the `transfer` at line 53 is true that depending on the order of execution, some joins executed before or after it could alter the transferred amount, even if it is not a major issue for the contract purpose; for line 47 it does not seem to be a correct warning: the transferred amount is equal to the `remainder`, which depends only on the `msg.value`

3. 🟧 *Reentrancy with constant gas* at line 38: the warning is due to the `transfer` in the for; following the *Checks-effects-interactions* pattern all transfers should be postponed after the state update, it would have been possible to do it here, even if the code would have become a bit more cumbersome. In practice there shouldn't be problem even in this way, considering that even in the case of a call of the `join`, the user has to pay 200 finney to enter the queue, so it cannot really take an advantage in doing this. It should also be taken into consideration the limit of 2300 gas which redcues a lot the possible actions.

4. 🟥 *Unrestricted write to storage* at lines 34, 37 and 60: for lines 34 and 37 the write is controlled by the `join` method, it's not really "free", while there is no write at line 60, so it's not clear what it is referred to

5. 🟧 *Division Before Multiplication* at lines 42 and 45: for line 42 it is not clear to what is the division referred to, clearly there is no division of `memberQueue.lenght`, but maybe it is due the check of the parity at line 36, anyway the problem is not present; for line 45 the warning makes sense since `entries` has been divided by 100 finney at line 30, but the algorithm takes this into consideration and it is correct to have the truncate division result in `entries` and then multiply it.

6. 🟧 *Division influences Transfer Amount* at lines 38 and 47: at line 38 there is no real division, maybe is again due to the parity check at line 36, anyway not a real problem; the warning is correct about line 47 but it's exactly what is taken into consideration in this phase of the method, where the remainder of the division is sent back to the user

7. 🟥 *Missing Input Validation* at line 56: given the purpose of the function and the `onlymaster` modifier, there is no reason to check in any way the parameter in input

8. 🟥🟧 *Unrestricted ether flow* at line 38 and 47: the `transfer` at line 38 and 47 are under the control of the algorithm

9. 🟧 *Unsafe Call to Untrusted Contract* at lines 47 and 53: at line 47 the user to which Ether are transferred to is the `msg.sender`, it's in the purpose of the method, while at line 53 it's the `master` which can be consider for sure realiable.

We did the same with the code of *WETH9* :

1. 🟧 *Unrestricted write to storage* at lines 55 and 56: they correspond to increment and decrement of the `balanceOf` of a quantity equal to the `wad` parameter. Given the `require` at line 48 they doesn't seem really a vulnerability.

2. ■ *Missing Input Validation* at lines 34, 40, 44: again the check in line 48 should protect from errors about the parameters of `transfer` and `transferFrom`. For the `wad` parameter of the `approval` method, given it's `uint` type and again the check of line 48, every value of it should not cause problems. This means we can allow another user to spend more than our actual balance, but it's only a necessary and not a sufficient condition for him to physically transfer this money out of our balance, he can transfer only what we have, so it's not a real problem.

3. ■ *Unrestricted ether flow* at line 26: here the `msg.sender` can withdraw money that he owns, so the control on the transfer is implicit by the use of `msg.sender`

4. ■ *Unsafe Call to Untrusted Contract* at line 26: it's the same line of the previous alert, but here the problem is different. Considering that there is no code, except for the event, after this line in the `withdraw` method, this is safe to reentrancy attacks. The problem can be another and it is related to the use of `transfer` function. This has been considered the standard for years to avoid reentrancy attacks, given that it allows only to consume 2300 gas in the fallback function of the recipient's contract, but considered the change of the gas cost of some instructions, which happened and could happen again in the future, it is not anymore considered the best way to transfer ETH. More details can be found in [32]. Considering that this contract is online since December 2017, it is not something that could have been expected by the authors, and here the problems of immutability of contracts' code is evident.

6.3.2 Evaluation

We have verified that most of the warnings reported by Securify were not actually real problems, except for transaction order problems of *Pyramid*, which for sure should be taken into consideration. The amount of false positives is due to the conservative approach of the tool, which is appreciable given the context. Only a few of the reported issues seemed completely unreasonable. The ease of use of the online scanner make it useful for every smart contract developer, even if it is something different than the complete formal verification, he's just focused on security patterns, which anyway have been the cause of big attacks to smart contracts.

6.4 Coloured Petri Networks

6.4.1 Model

The modelling methodology adopted in designing the following CPNs builds upon the one showed in [20] where a far simpler smart contract than the ones we are considering was taken into account.

First and foremost the relative complexity of the smart contracts called for an analysis of the abstraction mechanisms provided by CPN tools. CPN tools uses an extension of SML called CPN-ML for defining colors (called colorsets in the tool) while it also allows defining SML functions to be executed on edges so as to make computations over the markings traversing such edges. CPN-ML is thus used for defining data types local to the smart contract and all the helper types needed for more expressive modelling as in listing 5 .

After having defined the types one must consider what computations to encode with the CPN itself and what instead to handle with SML functions executed on the edges, the consequences of such choice have been explored through the Pyramid contract of which two models were devised, the PyramidHybrid and the PyramidFull.

```
colset Finney = INT;

colset Address = STRING;

colset AddressList = list Address;

colset Pyramid = record
master: Address *
memberQueue: AddressList *
queueFront: INT;

colset Account = record
balance: REAL *
id: Address;
```

Listing 5: Color definition for Pyramid

Pyramid hybrid and full are mostly equal for what concerns their colors, they have the same definitions except for one which in the former is used to encode the result of the loop while in the latter is used to encode the values updating throughout the loop. Speaking of the CPNs themselves they are exactly the same except for the section which encodes the loop, in the hybrid case the loop is completely encoded in an SML function computed on an edge as in listing 6 while in the full case no functions are specified and the loop is encoded in a substitution transition showed in figure 7.

The implicit component based approach in CPN modelling allows for an almost seamless implementation of both the hybrid and the full approach as we can see in figure 8 where the only difference lies in the transition encoding the loop which in one case is an actual transition while in the other it is a substitution transition condensing the net in figure 7.

Concluding on the Pyramid contract the top level net represents its interface. In Pyramid the top level is not actually identical to the contract's interface because the transfer to members of the pyramid needed to be abstracted away with the helper place Transfer targets (see figure 9) in order not to increase complexity while still keeping track of the transfers made inside the loop of the join function.

```
fun updateQueue (memberQueue: AddressList, queueFront: INT,
                 sender: Address, entries: INT,
                 transferTargets: AddressList) =
    let val updatedMemberQueue = rev (sender::rev memberQueue) in
    if  entries > 0 then
        if  List.length (updatedMemberQueue) mod 2 = 1 then
            updateQueue(updatedMemberQueue,
                        queueFront + 1, sender,  entries -1,
                        List.nth (memberQueue,
                                  queueFront)::transferTargets)
        else
            updateQueue(updatedMemberQueue, queueFront, sender,
                        entries -1, transferTargets)
    else
        {memberQueue_loop = memberQueue,
         transferTargets_loop = transferTargets}
end
```

Listing 6: SML function encoding Pyramid loop

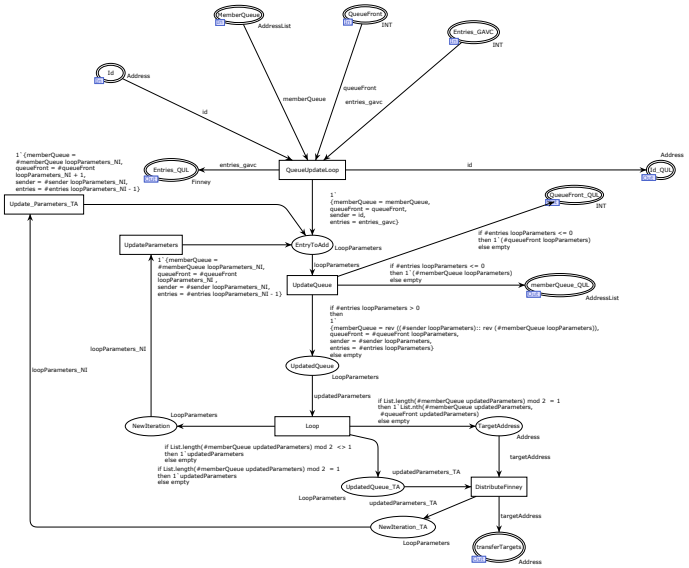
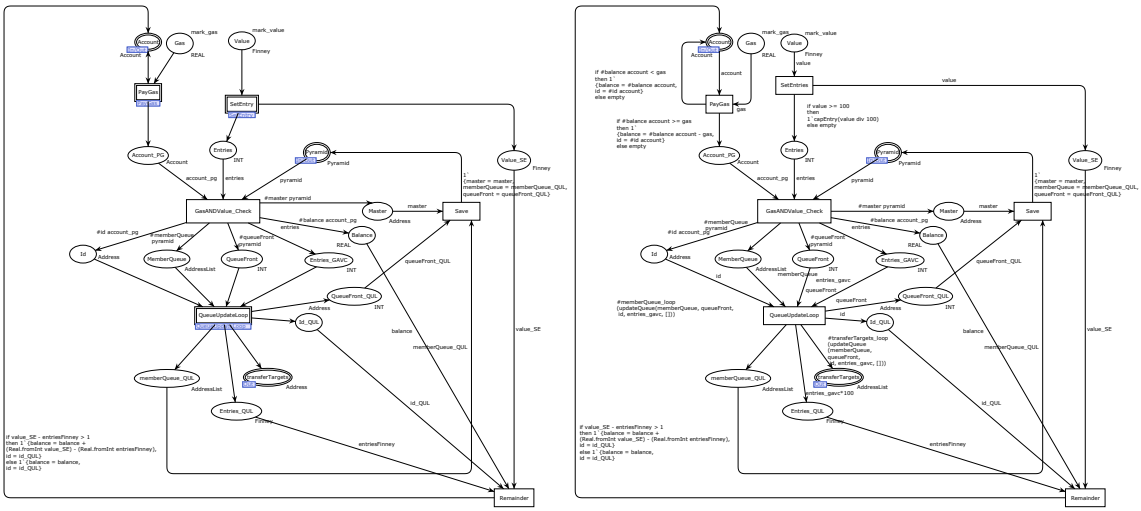
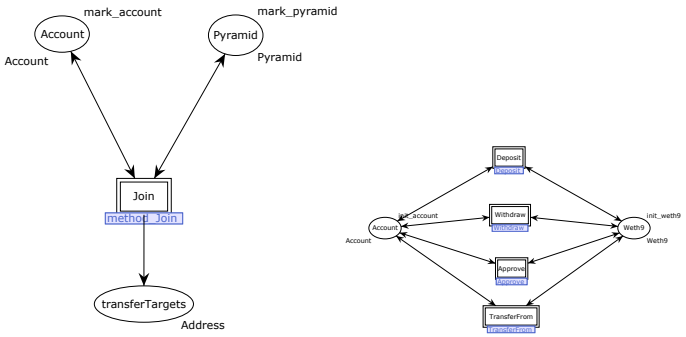


Figure 7: Pyramid loop implementation



(a) PyramidFull join method (b) PyramidHybrid join method

Figure 8: Join method model with Full and Hybrid approach



(a) Pyramid top-level (b) WETH9 top-level

Figure 9: Top level of Pyramid and WETH9

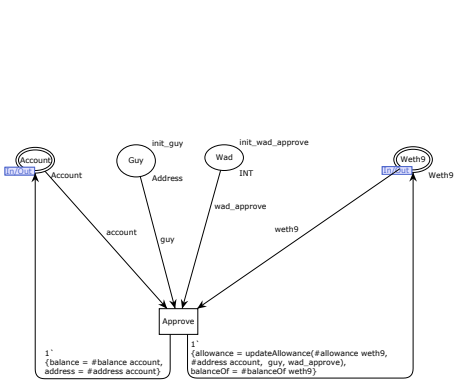
Given all the previous considerations, when modelling WETH9 the hybrid approach was mandatory in order to avoid the explosion of the state space. Hybrid modelling allowed to highlight further the importance of substitution transitions in modelling every method distinctly and clearly defining the contract interface throughout the top level net in figure 9

As we can see the CPN implementation is far simpler (see 10), the most complex sml methods (see for example 7) are far simpler than the ones outlined before. This simplification is mainly due to the fact that the methods are simpler than the ones considered for Pyramid. As showed in figure 9 while the methods of WETH9 are simpler the interface in general is larger than the one of the Pyramid contract.

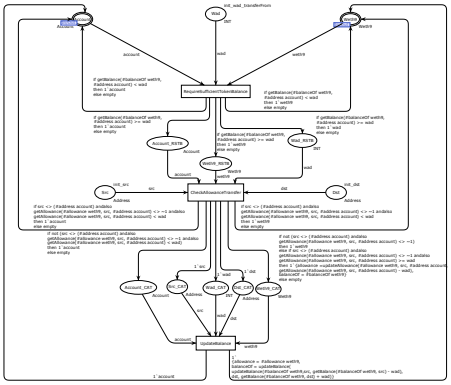
```
fun updateAllowanceElement (allowanceElement: AllowanceElement,
                            guy: Address, sender: Address, wad: INT) =
    {allower = sender, allowed = (List.map(fn allowedElement =>
        if (#address allowedElement) = guy then
            {address = #address allowedElement, value = wad}
        else
            allowedElement) (#allowed allowanceElement))}

fun  updateAllowance (allowance: Allowance, sender: Address, guy: Address, wad: INT)
     =
     List.map(fn allowanceElement =>
                if (#allower allowanceElement) = sender then
                    updateAllowanceElement(allowanceElement, guy, sender, wad) else
                allowanceElement)
              allowance
```

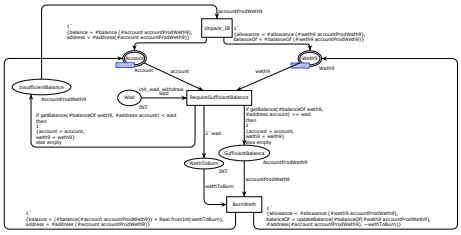
Listing 7: Example SML functions in WETH9



(a) WETH9 approve method



(b) WETH9 transferFrom method



(c) WETH9 withdraw method

Figure 10: WETH9 methods models

6.4.2 Verification

CPN model verification begins by defining an initial marking over which the CPN model behavior evolves. CPN tools allows several analysis, we will take into consideration state space analysis. CPN simulation may be useful for debugging the model in the modelling stage, which as shown earlier may not be trivial, but does not actually hold any significance in proving system properties as it works on a specific input set. Another tool that will not be taken into consideration is ASK-CTL a branching time logic which is used to encode rules to be checked over the state space. Again, the validity of such rules is limited to the specific state space taken into consideration and not to the model itself. Both the approaches not taken into consideration may actually be used if a suitable state space is given, which is what we are addressing next.
The state space of a CPN depends on the initial marking which determines which transitions fire and which do not, called dead transitions. We do note that for a state space to be meaningful no dead transitions should exist otherwise it would mean that some branches in the execution flow of the program are not taken, in case no branches are hidden inside SML methods.

```
Statistics


 State Space
    Nodes:  762
    Arcs:   1718
    Secs:   0
    Status: Full

 Scc Graph
    Nodes:  762
    Arcs:   1718
    Secs:   0


Home Properties


 Home Markings
    [762]


Liveness Properties


 Dead Markings
    [762]

 Dead Transition Instances
    None

 Live Transition Instances
    None


Fairness Properties

    No infinite occurrence sequences
        .
```

Listing 8: PyramidFull state space

```
Statistics


 State Space
    Nodes:  9
    Arcs:   10
    Secs:   0
    Status: Full

 Scc Graph
    Nodes:  9
    Arcs:   10
    Secs:   0


Home Properties


 Home Markings
    [9]


Liveness Properties


 Dead Markings
    [9]

 Dead Transition Instances
    None

 Live Transition Instances
    None


Fairness Properties

    No infinite occurrence sequences
        .
```

Listing 9: PyramidHybrid state space

In fact when moving from a Full approach to a Hybrid approach, as previously noted, we move from a larger to a smaller state space (see state space reports 8, 9) but we hide some branches inside methods and so the absence of dead transitions does not suffice anymore for saying that all branches are taken given an initial marking. Even when considering full approaches when the model complexity grows building an initial marking that fires every transition becomes non trival, especially when dealing with non trivial data structures like the allowance in WETH9 (see allowance in listing 2).

The state of a CPN is encoded through the set of all places and the markings they contain thus the state space is made of all the possible configurations of markings over places after transitions fired. We must now inspect the relevance that dead markings and home markings have in mapping state space properties to software properties. Dead markings are markings from which no other marking is reachable that is they are sink states. Home markings instead are markings which are reachable from every marking in the state space. When a marking is both a dead marking and a

home marking we know that it works as a sink for the whole space as it is for Pyramid in 8, 9. From the software point of view the fact that only one dead marking exists which is also a home marking assures that no deadlocks are reachable with the given input, as no information stays trapped inside the state space without reaching the sink state. Moreover the existence of exactly one home marking which is also a dead marking represents the fact that for the given input the software may reach one and only final state for every possible concurrent execution. In weth9 the interface of the contract being more complex does not produce home markings but produces 4 dead markings (see state space report 10), in this case the state space is harder to inspect because it is larger than the hybrid pyramid state space and far harder to reason about.

```
Statistics


 State Space
    Nodes:  42
    Arcs:   58
    Secs:   0
    Status: Full

 Scc Graph
    Nodes:  42
    Arcs:   58
    Secs:   0


Home Properties


 Home Markings
    None


Liveness Properties


 Dead Markings
    [31,32,41,42]

 Dead Transition Instances
    None

 Live Transition Instances
    None


Fairness Properties

    No infinite occurrence sequences
        .
```

Listing 10: Weth9 state space

```
val init_dst = "Guattari"
val init_src = "Deleuze"
val init_wad_transferFrom = 30
val init_wad_withdraw = 10
val init_account = 1`{balance =
    100.0, address = "Deleuze"}
val init_weth9 = 1`{allowance = [
{allower = "Guattari",  allowed = [{
    address = "Deleuze", value =
    50}]},
{allower = "Deleuze", allowed = [{
    address = "Guattari", value =
    0}]}],
balanceOf = [{address = "Deleuze",
    balance = 30},
{address = "Guattari", balance =
    100}]}
val init_depositValue = 20
val init_guy = "Guattari"
val init_wad_approve = 10
```

Listing 11: WETH9 initial marking

In general we may say that there exist 4 possible termination states for every concurrent run if this is always made by the same user, as it is the case in weth9 where in the initial marking (see listing 11) only "Deleuze" is considered as the account calling the methods. It is possible to further inspect the 4 final states to understand the 4 possible final configurations but this may be difficult

(while theoretically possible) given the complexity of the markings involved.

The boundedness report (contained in the state space reports and here omitted for brevity) does also hold verification relevance as it highlights how many tokens each place may hold at most thus encoding how many operations one user can do simultaneously. In the pyramid case some places could not hold single markings in the initial state as multiple operations were needed to make all transitions fire (some branches were mutually exclusive) but in general it is easy to see that all places which did not contain initial markings, both for WETH9 and for Pyramid, could not contain more than one marking thus demonstrating that it was impossible for a user to do more than one operation at a time. Initial markings for Pyramid, common to Full and Hybrid approach, are here omitted because they are totally arbitrary and can be built following exactly the example provided for weth9.

6.4.3 Evaluation

What arises from the previous CPN analysis is an approach which needs much closer examination before becoming of any practical relevance.
From the modelling point of view a clear methodology for smart contract modelling needs to be built in order to encompass non trivial cases as the ones examined here. In particular the relevance of SML functions as an abstraction mechanism needs to be clearly established as well as its influence on the verification process. On the other hand substitution transitions emerged as a useful tool for component based modelling of smart contracts in terms of their interfaces and subsequent implementations.
From the verification point of view more work needs to be done in mapping model properties to program properties. Specifically state space analysis must be inspected in a more systematic way as manual state graph exploration approaches are not feasible whene analysing non-toy smart contracts. For what concerns model checking through ASK-CTL properties it may become a useful tool only if a systematic way of generating initial markings is devised. Generating initial markings may be seen as exploring the space of all possible state spaces which from the software point of view corresponds both to changing the inputs and the behaviour of a number of interacting users. When the meaning of a marking is made clear, which is not always easy in real smart contracts, model checking may actually be used to prove relevant properties in specific scenarios.
After a clear understanding of the above, more complex models for the blockchain, in terms of gas expenditure and such, may be implemented through substitution transitions. Attacker models would also be susbstitution transitions encoding an attacker behavior when interacting with the contract, this would also mean that careful thought about attack scenarios has to be done, bridging to many other research fields, from behavioural sciences to computer security.

7 Discussion

The first question which emerged during our survey is: what do we really need to verify? The answer is not unique and we can distinguish between two levels.

On the first level we have *traditional* program verification, which has been applied for decades on many programming languages. Program verification holds peculiar relevance in smart contracts considering the impossibility to patch a smart contract once it has been deployed on the blockchain.

On the second level we have the blockchain, with all the elements implied by its architecture: ether transfers, transactions order, gas consumption and so on. What role should the blockchain

take in the verification process? Ignoring it is not an option, as we have seen many severe bugs arise due to infrastructural blockchain charachteristics and the interactions smart contracts have with them. We have analyzed tools completely focused on the blockchain typical bugs (*Securify, Zeus, EthRacer*), but they don't offer the possibility of a wide personalized verification of other properties.

On the other side we have more traditional approaches, both model-checking or theorem-proving oriented, which are able to verify properties such as pre-post conditions on methods, assertions and liveness properties related to the state of the contract. We have to say that none of these tools seem ready for production. Some of them lack in the supported syntax of Solidity (*F*, Scilla*), others lacks in the definition of the translation (*Why3).* Model checking approaches present performance problems, as we have seen with *Promela*, together with a complex modelling phase which severely impacts the subsequent verification phase, see the *CPN* models for example.

Considering the fact that some of these approaches do not consider the entire blockchain behaviour we should also ask ourselves if we can trust their results. If we consider the Promela approach we can see that its completely interleaved process execution should somehow protect us against transaction ordering problems, but what about reentrancy bugs? Considering how we have modelled the transfer of ether, fallback functions are ignored. Also the asynchronous interactions with external oracles seem difficult to be properly modelled and the gas consumption is not considered at all. Other model checking approaches, such as *BIP*, promise to take into account blockchain models without ever mentioning how this would affect the state space. In the *CPN* models, for example, more sophisticated gas policies may be implemented but without serious consideration of the available abstraction mechanisms such additions may render the models impossible to verify.

The K framework is based on a precise definition of the syntax and semantic of EVM and it is thus able to verify with precision properties related to the contract state, the gas consumption and even the events logging. But, as it is stated in [11], it is still unable to identify typical antipatterns that lead to important bugs in the past.

The MDP approach outlined in [4] leverages results from Game Theory highlighting the central role which interaction plays when it comes to smart contract modelling. Interactions are very hard to model clearly when complex systems are taken into account: smart contracts interacting with each other, users taking advantage of smart contract enabled services, attackers looking after huge amount of money and miners verifying transactions in real-time, just to name some of the actors involved. The role played by game theory in the aforementioned approach is yet too marginal but it may show an interesting path ahead.

8 Conclusion

In the end we have to say that today a safe and complete way to formally verify a smart contract doesn't seem to exist. The best that can be done is the use of a combination of these tools, depending also on the type of properties which need to be verified and the complexity of the smart contract. For sure the use of tools such as *Securify* is always advisable: even if they have a quiet large number of false positive warnings, it's not hard to go through them and identify the real problems.

9 Future work

The analysis performed here can be expanded with the empirical test of some of the available but not tested approaches and it can be kept updated with the new emerging approaches in the field, the future improvements of the analyzed approaches and also the evolution of Ethereum, Solidity and in general of the smart contracts area, which may have immediate and relevant impact on their formal verification techniques.

References

[1] Tesnim Abdellatif and Kei-Leo Brousmiche. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, February 2018.

[2] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: Identification, analysis, and impact. *Future Generation Computer Systems*, 08 2019.

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016.

[4] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods. In *Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security - Volume 9465*, pages 142–161, Berlin, Heidelberg, August 2015. Springer-Verlag.

[5] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120, Vienna, Austria, September 2001. Association for Computing Machinery.

[6] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Assume-Guarantee Verification for Interface Automata. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods*, Lecture Notes in Computer Science, pages 116–131, Berlin, Heidelberg, 2008. Springer.

[7] Ethereum Foundation. CRITICAL UPDATE Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/. Library Catalog: blog.ethereum.org.

[8] Ethereum Foundation. DAOs, DACs, DAs and More: An Incomplete Terminology Guide. https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide/. Library Catalog: blog.ethereum.org.

[9] Parity team publishes postmortem on $160 million ether freeze. `https://www.coindesk.com/parity-team-publishes-postmortem-160-million-ether-freeze`.

[10] Dominik Harz and William J. Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *ArXiv*, abs/1809.09805, 2018.

[11] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium*, 2017.

[12] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[13] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *Int J Softw Tools Technol Transfer*, 9(3):213–254, June 2007.

[14] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.

[15] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[16] Lars Michael Kristensen and Michael Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In Stefan Kowalewski and Marco Roveri, editors, *Formal Methods for Industrial Critical Systems*, Lecture Notes in Computer Science, pages 215–230, Berlin, Heidelberg, 2010. Springer.

[17] Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic Model Checking: Advances and Applications. In Rolf Drechsler, editor, *Formal System Verification: State-of the-Art and Future Trends*, pages 73–121. Springer International Publishing, Cham, 2018.

[18] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical Model Checking: An Overview. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, Lecture Notes in Computer Science, pages 122–135, Berlin, Heidelberg, 2010. Springer.

[19] J. Liu and Z. Liu. A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904, 2019.

[20] Zhentian Liu and Jing Liu. Formal Verification of Blockchain Smart Contract Based on Colored Petri Net Models. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 555–560, July 2019.

[21] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Gabor Madl, Luis Bathen, German Flores, and Divyesh Jadav. Formal Verification of Smart Contracts Using Interface Automata. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 556–563, July 2019.

[23] Kjeld H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In Mogens Nielsen and Dan Simpson, editors, *Application and Theory of Petri Nets 2000*, Lecture Notes in Computer Science, pages 367–386, Berlin, Heidelberg, 2000. Springer.

[24] Yvonne Murray and David A. Anisi. Survey of formal verification methods for smart contracts on blockchain. *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6, 2019.

[25] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. *no journal*, page 9, 2008.

[26] Zeinab Nehai and François Bobot. Deductive proof of ethereum smart contracts using why3. *ArXiv*, abs/1904.11281, 2019.

[27] Thomas Osterland and Thomas Rose. Model checking smart contracts for ethereum. *Pervasive and Mobile Computing*, 63:101129, 02 2020.

[28] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 912–915, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Pyramid smart contract source code on etherscan. `https://etherscan.io/address/0xb978A5F4854274bc5196bC2a4633863CB3A0a6b7#code`.

[30] Grigore Rosu. Formal design, implementation and verification of blockchain languages (invited talk). In *FSCD*, 2018.

[31] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *ArXiv*, abs/1801.00687, 2018.

[32] Stop using solidity's transfer now. `https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/`.

[33] Andrei Stefanescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Rosu. Semantics-based program verifiers for all languages. In *OOPSLA 2016*, 2016.

[34] Nick Szabo. Smart Contracts. `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`.

[35] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Michael Westergaard and H.M.W. (Eric) Verbeek. CPN Tools – A tool for editing, simulating, and analyzing Colored Petri nets. Library Catalog: cpntools.org.

[37] Weth9 smart contract source code on etherscan. `https://etherscan.io/address/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2#code`.