



Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach

Ikram Garfatta

University of Tunis El Manar, National Engineering School
of Tunis, OASIS
Tunis, Tunisia
University Sorbonne Paris North, LIPN UMR CNRS 7030
Villetaneuse, France
ikram.garfatta@lipn.univ-paris13.fr

Mohamed Graïet

University of Monastir, Higher Institute for Computer
Science and Mathematics
Monastir, Tunisia

Kaïs Klai

University Sorbonne Paris North, LIPN UMR CNRS 7030
Villetaneuse, France
kais.klai@lipn.univ-paris13.fr

Walid Gaaloul

Institut Mines-Télécom, Télécom SudParis, SAMOVAR
UMR 5157
Évry, France

ABSTRACT

Despite the benefits that the Blockchain technology brings to many application fields, its adoption does not come without challenges. Smart contracts, which are at the core of 2nd generation blockchains, can often be riddled with vulnerabilities that can be exploited to attack the platform and threaten its security. It is therefore crucial for the protection of the designed systems to prove the correctness of the smart contracts to be deployed. Approaches have been proposed to detect generic vulnerabilities like reentrancy, but the results would often include false positives where the detected bug is either non-existent or not exploitable. Besides, such approaches do not offer to check contract-specific properties. The work presented in this paper is situated as part of a formal approach that we have proposed in an attempt to bridge this gap. This previously outlined approach is based on the transformation of Solidity smart contracts into Coloured Petri nets, which provides the possibility to verify smart contracts with reference to properties expressed as Linear Temporal Logic (LTL) formulae. Herein we extend our previous work on mainly two levels: first, by taking into account the concept of *function calls* in the transformation and second, by focusing on the LTL properties that can define the correctness of a smart contract. Such properties can be specific to the control- or data-flow of the contracts being checked. They can also be used to express vulnerabilities as we showcase by proposing LTL formalizations for six vulnerabilities from the literature. We then leverage the capability of the *Helena* model checker to detect these vulnerabilities while discerning their exploitability, as well as check temporal-based contract-specific properties.

CCS CONCEPTS

• **Security and privacy** → **Formal methods and theory of security**; • **Software and its engineering** → **Model checking**; **Formal software verification**;

KEYWORDS

Blockchain, Model checker, Smart Contract, Solidity, Coloured Petri Nets, Temporal properties

ACM Reference Format:

Ikram Garfatta, Kaïs Klai, Mohamed Graïet, and Walid Gaaloul. 2022. Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*, April 25–29, 2022, Virtual Event, . ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3477314.3507309>

1 INTRODUCTION

First known as the supporting technology of the renowned Bitcoin cryptocurrency, the Blockchain has ever since known many advances that took it from being merely a database recording transactions between parties to being a computational platform on which smart contracts can be invoked as transactions. This leap significantly expands the power of blockchain systems, and increases their reach to many application fields. This can be particularly observed in the growing interest blockchains are gaining as part of IT systems, in domains such as health records, banking, voting, personal identity, etc [26].

While blockchain technology itself has proved to be highly-tamper resistant, many attacks with significant consequences have been waged on several blockchain platforms, exploiting hidden vulnerabilities in deployed smart contracts and exposing the defectiveness of the targeted applications. In 2010, 92 billion BTC were generated out of thin air by exploiting an integer vulnerability on the Bitcoin blockchain. One of the most infamous attacks on Ethereum was the one exploiting a reentrancy vulnerability in the DAO and resulting in 3.6M of stolen Ether. A vulnerable blockchain-based application does not have to be the target of a malicious attack to malfunction. For instance, the Parity multisig wallet was subject

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SAC '22, April 25–29, 2022, Virtual Event,

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-8713-2/22/04...\$15.00

<https://doi.org/10.1145/3477314.3507309>

to an unintentional accident caused by a self-destruct vulnerability in 2017 and resulting in freezing 500K of Ether.

Given the importance of the assets circulating in each blockchain, securing the applications deployed on such distributed ledgers is considered mission-critical, and seeing that most of the vulnerabilities allowing the breaches are rooted in the smart contracts of the applications, their verification before deployment is crucial.

Informal as well as formal methods have been proposed to enhance the security of smart contracts and ensure their correctness. While informal techniques can test a smart contract under certain scenarios, they cannot be relied on to verify specific properties defining its correctness (e.g., absence of integer overflow vulnerabilities, deadlock-freedom) which is where formal techniques prove to be efficient. We note that we are interested in Ethereum as it is currently the second largest cryptocurrency platform after Bitcoin besides being the inaugurator of smart contracts, and more particularly in Solidity [1], the most popular language used by Ethereum. We also note that while Ethereum allows smart contracts to be written in a ‘Turing complete’ language that facilitates semantically richer applications than Bitcoin which allows very simple forms of smart contracts, the former also enlarges the threat surface, as evidenced by the many high-profile attacks.

In this paper, we build on our proposed approach based on Coloured Petri Nets (CPNs) [14], for the formal verification of Solidity contracts. Our choice of this formalism is driven by its ability to combine the analysis power of Petri nets with the expressive power of programming languages, which makes it suitable for the modeling and verification of large and complex systems. CPNs have in fact been leveraged in various contexts in literature [10, 23] proving their efficiency for formal verification. The cornerstone of our approach was first set in [11] where we presented a rough outline of the verification method that we propose and a preliminary experimentation using two different model checkers. In [12], we provided more details on the patterns we propose for the transformation of a Solidity smart contract into a hierarchical CPN model depicting the functionality of the former. In this present paper, we propose improvements on two levels:

- (1) first we refine our proposed transformation by taking into account the concept of *function calls*. Having initially considered the basic concepts of Solidity in [12] that allow the verification of a single smart contract whose functions are invoked by external users, we now focus on supporting the verification of functions that can also be invoked by other functions (either from the same contract or other contracts), the main implication of which being the added support for the verification of multiple interacting smart contracts;
- (2) and second, we propose a formalization of a set of vulnerabilities as LTL formulae. The correctness of the represented contracts is then proven by analyzing the generated CPN model and verifying it w.r.t temporal properties that can be either predefined for vulnerabilities or other contract-specific properties relevant to the contract’s data- and control-flows, which gives the designer a wide range of control to define the correctness of the contract.

We propose an algorithm that automates our transformation, implement a prototype to prove its feasibility and leverage the *Helena* [8] tool to verify system properties.

The remainder of this paper is organized as follows: Section 2 provides essential prerequisites on CPN and LTL. Section 3 presents the related works. In Section 4, we present use cases to introduce the considered vulnerabilities. We revisit our transformation approach in Section 5 to include *function calls* and dedicate Section 6 for the formalization of a selection of vulnerabilities using LTL as well as providing details on the application of the whole verification approach. Section 7 concludes the paper and outlines some future perspectives.

2 BACKGROUND

2.1 Coloured Petri Nets

A Petri net (PN) [22] is a formal model with mathematics-based execution semantics. It is a directed bipartite graph with two types of nodes: places (drawn as circles) and transitions (drawn as rectangles). Despite its efficiency in modelling and analysing systems, a basic PN falls short when the system is too complex, especially when data representation is required. To overcome such limitations, extensions to basic PN were proposed, equipping the tokens with colours or types [13], [28] and hence allowing them to hold values. A large PN model can therefore be represented in a much more compact and manageable manner using a *Coloured Petri net*.

A Coloured Petri Net [14] combines the capabilities of Petri nets, from which its graphical notation is derived, with those of CPN ML, a functional programming language based on Standard ML [21], to define data types. The formal definition of a CPN is given in Definition 2.1 and the main concepts needed to define its dynamics are given in Definition 2.2.

Definition 2.1 (Coloured Petri net [14]). A Coloured Petri Net is a nine-tuple $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, where:

- (1) P is a finite set of *places*.
- (2) T is a finite set of *transitions* such that $P \cap T = \emptyset$.
- (3) $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed *arcs*.
- (4) Σ is a finite set of non-empty *colour sets*.
- (5) V is a finite set of *typed variables* such that $Type[v] \in \Sigma$ for all variables $v \in V$.
- (6) $C : P \rightarrow \Sigma$ is a *colour set function* that assigns a colour set to each place.
- (7) $G : T \rightarrow EXPR_V$, where $EXPR_V$ is the set of expressions provided by CPN ML with variables in V , is a *guard function* that assigns a guard to each transition t such that $Type[G(t)] = Bool$.
- (8) $E : A \rightarrow EXPR_V$ is an *arc expression function* that assigns an arc expression to each arc a such that $Type[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a (i.e., the type of the arc expression is a multiset type over the colour set of the connected place).
- (9) $I : P \rightarrow EXPR_{\emptyset}$ is an *initialisation function* that assigns an initialisation expression to each place p s.t. $Type[I(p)] = C(p)_{MS}$.

Definition 2.2 (CPN concepts [14]). For a $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, we define the following concepts:

- (1) $\bullet p$ and $p \bullet$ respectively denote the sets of input and output transitions of a place p .
- (2) $\bullet t$ and $t \bullet$ respectively denote the sets of input and output places of a transition t .
- (3) A *marking* is a function M that maps each place $p \in P$ into a multiset of tokens $M(p) \in C(p)_{MS}$.
- (4) The *initial marking* M_0 is defined by $M_0(p) = I(p)\langle \rangle$ for all $p \in P$.
- (5) The *variables of a transition* t are denoted by $Var(t) \subseteq V$ and consist of the free variables appearing in its guard and in the arc expressions of its connected arcs.
- (6) A *binding* of a transition t is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type[v]$. It is written as $\langle var_1 = val_1, \dots, var_n = val_n \rangle$. The set of all bindings for a transition t is denoted $B(t)$.
- (7) A *binding element* is a pair (t, b) such that $t \in T$ and $b \in B(t)$. The set of all binding elements $BE(t)$ for a transition t is defined by $BE(t) = \{(t, b) | b \in B(t)\}$. The set of all binding elements in a CPN is denoted BE .
- (8) A *step* $Y \in BE_{MS}$ is a non-empty, finite multiset of binding elements.

A transition is said to be *enabled* if a binding of the variables appearing in the surrounding arc inscriptions exists such that the inscription on each input arc evaluates to a multiset of token colours that is present on the corresponding input place. *Firing* a transition consists in removing (resp. adding), from each input place (resp. to each output place), the multiset of tokens corresponding to the input (resp. output) arc inscription. For more details on the CPN formalism and the formal definition of its semantics, we refer readers to [14].

CPN example. To better explain the basic concepts of CPN, we use the simple CPN model of Fig. 1. *Couple_Type* is defined as the product of two integers and *Triplet_Type* as the product of three integers. x and y are two integer variables. In a CPN model, each place has a colour that determines the kind of data it can contain. We say that $p1$ is of colour (or type) *Couple_Type* and $p2$ is of colour *Triplet_Type*. Initially, the place $p1$ contains three tokens with different values (three different couples). The expressions on the arcs have to correspond to the colours of their respective places (e.g., the expression on the outgoing arc of $p1$ has to conform to its colour *Couple_Type*). In this CPN, (x, y) can be bound to any of the tokens in $p1$. For example, if it is bound to the first token $(2, 5)$, the firing of transition $t1$ results in removing that token from $p1$ and adding a token with the value $(2, 5, 7)$ to $p2$.

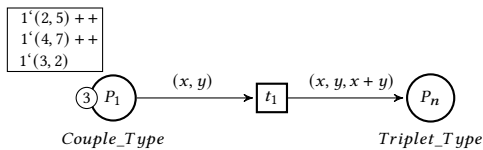


Figure 1: A simple example of CPN

2.2 Linear Temporal Logic

The approach presented in this paper is primarily based on model checking of CPN models w.r.t formulae expressed in Linear Temporal Logic (LTL). This logic was first introduced in [24] as a means to reason about concurrent programs.

In LTL, a classical timeline that starts “now” is considered, where every moment has a unique possible future. In other words, a model of LTL is an infinite sequence of indexed states ($i = 0, 1, 2, \dots$) where each point in time has a unique successor. An LTL formula is evaluated over such a sequence of states starting from an i ’th state. It contains a finite set *Prop* of atomic propositions, the usual Boolean operators \neg, \wedge, \vee , and \rightarrow , in addition to temporal operators:

- Until (\mathcal{U}): $\varphi \mathcal{U} \psi$ is true if ψ is true *now* or φ is true *now* and remains so until ψ holds.
- Next (\mathcal{X} or \bigcirc): $\mathcal{X} \varphi$ is true if φ is true in the next step.
- Globally (\mathcal{G} or \Box): $\mathcal{G} \varphi$ is true if φ is true in every step.
- Future (\mathcal{F} or \Diamond): $\mathcal{F} \varphi$ is true if φ is true *now* or in some future time step.

Definition 2.3 (LTL formula). An LTL formula can be inductively defined as follows:

- $\forall p \in Prop, p$ is an LTL formula.
- If φ and ψ are two LTL formulae, then $\neg \varphi, \varphi \wedge \psi, \varphi \vee \psi, \varphi \rightarrow \psi, \varphi \mathcal{U} \psi, \mathcal{X} \varphi, \mathcal{G} \varphi$ and $\mathcal{F} \varphi$ are LTL formulae too.

The technique of model checking checks that a system, starting at a start state, satisfies a specification [25].

3 RELATED WORKS

There are two primary lines in existing studies on formal verification of smart contracts [9]. Theorem proving is the basis for the first group of works [2, 5]. The fundamental concept is to convert the contract’s code (typically its associated EVM bytecode) into a theorem prover-processable code and then use the latter to discharge proofs on the produced code’s correctness. The verification is not automatic in this situation, and the user’s competence is required to manage the prover and manually discharge proofs.

Model checking is used in the second category of research studies. The majority of these studies make use of symbolic model checking in combination with other techniques like symbolic execution [16] and abstraction [3]. Oyente [19], a tool that targets four vulnerabilities: transaction order dependence, timestamp dependence, mishandled exceptions, and reentrancy, is the first attempt in this category. It works at the contract’s EVM bytecode level, generating symbolic execution traces and analyzing them for the satisfaction of particular requirements on the paths, indicating the presence of vulnerabilities. Multiple tools followed the lead of this work, either by exploiting some of its components in their implementations, such as GASPER [6], which reuses Oyente’s generated control flow graph for the detection of bytecode patterns with high gas costs, or by extending it like in the case of Osiris [27], with the goal of supporting the detection of other vulnerabilities.

Zeus [15], which works on the source code of the smart contract, also uses symbolic model checking. The user must provide the criteria to be verified as a CHC-based policy written in XACML, according to which the code is instrumented before being translated into a low-level intermediate representation (LLVM bytecode), which

is then given to a verification engine.

VeriSolid [20] is an FSM-based method that, unlike the others, focuses on creating a correct-by-design contract rather than detecting bugs. The authors suggest to transform an FSM-based contract into Solidity code and allow the user to define expected behavior in the form of liveness, deadlock freedom, and safety properties that may be represented using templates for CTL properties and validated by a backend symbolic model checker.

Recently, attempts have been made to employ colored petri nets for smart contract verification. [18] presents an example of behavioural properties verification using a CPN model for a crowdfunding smart contract. It does not, however, provide a comprehensive and generic technique that can be used to any smart contract, since the work only presents a case study of a manual translation from the use case's contract to the CPN model without specifying general transformation rules. [7] proposed another CPN-based solution. The authors start with the bytecode and apply Hoare's logic to build a CPN model, which is subsequently used for the contract's security analysis. Although, it is based on CPN, this technique cannot be used to verify data-flow properties since the produced model emphasizes on the workflow retrieved from the contract's CFG.

The techniques based on symbolic execution (e.g., [6, 19, 27]) often employ under-approximation (e.g., in the form of loop limits) in order to create the traces that would be used for the verification, which implies that important violations might be ignored. This explains why their reported findings contain false negatives and/or positives. We also highlight the fact that the majority of existing research focuses on particular smart contract vulnerabilities, with just a few studies allowing for the expression of customisable features, which are often related to control flow-related aspects. None of these studies, in fact, focus on data-related properties. It's worth noting that, due to Solidity's lack of formal semantics, the majority of the presented techniques work with the EVM bytecode rather than the Solidity code. However, this often leads to loss of contextual information, therefore limiting the range of characteristics that may be checked as a result.

Our suggested solution seeks to address these issues by allowing developers to create behavioural and contract-specific properties (in the form of temporal formulae) that can be based on the contract's data flow and hence are not limited to a small number of known vulnerabilities. We note that in our work, the six vulnerabilities that we consider in Section 6.1 are given as mere examples to prove that the expressiveness of LTL formulae can cover vulnerabilities from the literature even though our focus is on the verification of contract-specific properties. Furthermore, we point out that our method uses an explicit model checking strategy and that our transformation process works on source code rather than bytecode. As a result, we avoid the repercussions of under-approximation as well as loss of contextual information.

4 VULNERABILITIES THROUGH USE CASES

A Solidity smart contract may look like a JavaScript or C program syntax-wise, but they are actually dissimilar since the underlying semantics of Solidity is different from traditional programs. This naturally calls on more vigilance from programmers who might be faced by unconventional security issues as vulnerabilities in smart

contracts seem to often stem from this gap between the semantics of Solidity and the intentions of the programmer [4].

In the following we present two use cases to explain the vulnerabilities treated in this work to provide a better understanding of how we verify them later. The full Solidity files can be found in our repository¹.

One of the most widespread smart contract applications is delivering gambling services. Thanks to Blockchain's decentralized nature and the transparency of its transactions, players can have a clear view of the behaviour of the game and are therefore led and incentivized to put their trust in the system which is determined by the rules implemented by its contracts. Our first Solidity example (Listing 1) is based on a published contract² implementing a lottery game. It has been tweaked to illustrate more vulnerabilities without altering its purpose. A player participates in this game by sending an amount of ether equal to the *TICKET_AMOUNT* through *playTicket()*, which is then added to the game's *pot*. The winner is determined based on a *random* value calculated using the block's timestamp and the *LottoLog* is updated accordingly to keep track of the winners. The winner then gets paid by calling *getPot()* and the game's host (*bank*) can start a new round of lotto using *RestartLotto()*. This contract may seem fair to inexperienced Solidity developers, but it actually presents multiple vulnerabilities as we will later explain.

```

1 contract EtherLotto {
2     address public bank;
3     struct GameRecord {address winner; uint amount;}
4     uint8 gameNum;
5     GameRecord[] LottoLog;...
6     function EtherLotto() {...}
7     function RestartLotto() {...}
8     function playTicket() payable {
9         require(msg.value == TICKET_AMOUNT);...
10        uint random = uint(sha3(block.timestamp)) % 2;
11        if (random == 0) {
12            GameRecord gr;...
13        }
14    }
15    function getPot() {
16        require(won == true);
17        if (msg.sender == LottoLog[gameNum].winner){
18            msg.sender.call.value(LottoLog[gameNum].
19                amount)("");
20            pot = 0;}}}
```

Listing 1: Solidity example: EtherLotto.sol

We consider a second example³ (Listing 2) to emphasize on the harmful effect the self-destruction vulnerability can have on a contract. It implements another gambling game whereby a player sends 1 ether to the contract by calling *play()* in hopes to be the one to hit a milestone. Once the game is over (i.e., the *finalMilestone* is reached) winners claim their rewards through *claimReward()*.

```

1 contract EtherMilestone {
2     uint public payoutMilestone1 = 6 ether;...
3     uint public finalMilestone = 20 ether;
4     uint public finalReward = 10 ether;
5     mapping(address => uint) redeemableEther;
6     function play() public payable {
7         require(msg.value == 1 ether);
```

¹<https://depot.lipn.univ-paris13.fr/garfatta/sol2cpn>

²<https://etherscan.io/address/0xa11e4ed59dc94e69612f3111942626ed513cb172>

³<https://gist.github.com/vasa-develop/415a17c709d804a4d351485cd1b7c981>

```

8      uint currentBalance = this.balance + msg.value;
9      require(currentBalance <= finalMilestone);
10     if (currentBalance == payoutMilestone1) ...
11     else if ...
12     else if (currentBalance == finalMilestone )
13         redeemableEther[msg.sender] += finalReward;
14     return;}
15     function claimReward() public {
16         require(this.balance == finalMilestone);
17         require(redeemableEther[msg.sender] > 0);
18         redeemableEther[msg.sender] = 0;
19         msg.sender.call.value(redeemableEther[msg.sender]
20             )(" ");}}

```

Listing 2: Solidity example: EtherMilestone.sol

```

1  contract MaliciousContract {
2      uint ticket;
3      EtherLotto el = EtherLotto(0xbf0061dc...);
4      EtherMilestone em = EtherMilestone(0xc50164dfa...);
5      function playLotto() {
6          ticket = msg.value;
7          el.playTicket.value(ticket)();
8          el.getPot();}
9      function playMilestone() { em.play.value(1)();}
10     function getRevenge ( ) { selfdestruct(em);}
11     function () payable { el.getPot();}

```

Listing 3: A malicious smart contract in Solidity

Integer Overflow/Underflow: due to Solidity’s lack of safeguards on mathematical operators, errors such as overflows and underflows may occur as a result of violation of value limitations of integers. The `uint8 gameNum` variable in the *EtherLotto* contract can be the source of such a vulnerability when the game exceeds 256 rounds. In fact, at the 257th round, and due to Solidity’s wrapping in two’s complement representation for integers, `gameNum` will be set to 0, causing data errors/overwriting into the critical `LottoLog` variable.

Reentrancy: the main idea behind it is that a function can be interrupted in the middle of its execution and then be safely called again before its initial call completes. Once the second call completes, the initial one resumes correct execution. The simplest example is when a smart contract uses a variable to keep track of balances and offers a withdraw function. A vulnerable contract would make a transfer of funds prior to updating the corresponding balance which an attacker can take advantage of by recursively calling this function and eventually draining the contract. This can be illustrated by a call to the function `playLotto()` with a value of 10 in the *MaliciousContract* (Listing 3) which would start by playing a ticket in the *EtherLotto* contract by invoking its `playTicket()` function and then attempting its `getPot()` function. In the instance where attacker’s ticket is a winning one and the contract holds more than twice the amount of the `pot` in that round, a reentrancy attack can happen. In fact, by sending the jackpot to the winner (Listing 1), the *EtherLotto* contract invokes the *fallback* function of the *MaliciousContract*, which is an unnamed function used to receive data or Ether. This is where the control flow is handed over to the latter contract whose *fallback* function recursively calls `getPot()`, which is allowed since the conditions are still valid, until the *EtherLotto* contract’s balance is less than the current `pot`’s amount.

Self-Destruction: the `selfdestruct(address)` function, when implemented in a contract, removes all bytecode from the contract’s address to render it inaccessible and sends all its ether to the specified

address. The latter can be another contract’s address, in which case, the ether transfer happens forcibly, regardless of the recipient’s code (i.e., without invoking its *fallback* function). Getting back to our second example *EtherMilestone*, we note the use of `this.balance` in `play()` and `claimReward()`. A player who missed a milestone, could vengefully send ether using `selfdestruct()` (e.g., function `getRevenge()` in *MaliciousContract*) as to push the contract’s balance above the `finalMilestone`, locking all of the contract’s ether and denying the winners who had already reached some milestones their rewards since `claimReward()` would revert.

Timestamp dependence: since the execution on a Blockchain needs to be deterministic for all the miners to get the same results and reach a consensus, users usually resort to block-related variables such as timestamp as a source of entropy. Sharing the same view on the Blockchain, miners would generate the same result, albeit being unpredictable. Even though this seems to be safe, it gives the miners a small room for manipulation given that they can choose a timestamp within a certain range for the new block, which gives them the possibility to tamper with the results and put some bias towards a certain user for example. Such a vulnerability can be exploited by any contract relying on a time constraint to determine its course of action. In our *EtherLotto* example, the function `playTicket()` is timestamp-dependent.

Skip Empty Literal: the source of this vulnerability is the way the encoder of the Solidity compiler treats the arguments in a function call. In fact, when a function call’s argument is an empty string literal, it affects the following arguments which are shifted to the right by 32 bytes. This results in a function call with corrupted data.

Uninitialized Storage Variable: Solidity stores state variables sequentially. So in *EtherLotto*, the variable `bank` is stored in slot 0. Since Solidity uses storage for complex data types like structs by default when declared as local variables, they become pointers to storage. Because `gr` is uninitialized (Listing 1), it would actually point to the same slot as `bank`. When setting `gr.winner` to the first winner’s address, this is effectively changing the address stored in `bank` to the winner’s, which results in an unexpected behaviour by this contract. In our example, we present this vulnerability as an error unintentionally introduced by the contract’s owner and unintentionally exploited by the first winner. It can, however, be intentionally injected in a contract’s code or intentionally exploited by a user, as is the case in the *OpenAddressLottery*⁴ honeypot.

5 A SOLIDITY-TO-CPN FORMAL VERIFICATION

Our verification approach comprises mainly two steps [11, 12]:

- (1) transforming the smart contract’s Solidity code into a CPN model
- (2) model checking of the generated CPN model with regard to an LTL property that can express: (i) a vulnerability in the code or (ii) a contract-specific property

The first step consists in generating a CPN submodel for each function of the contract to be verified (see Figure 2). These submodels (that we call *level-0 submodels*) represent the internal workflows of the functions and serve as building blocks for the final HCPN

⁴<https://etherscan.io/address/0x741f1923974464efd0aa70e77800ba5d9ed18902>

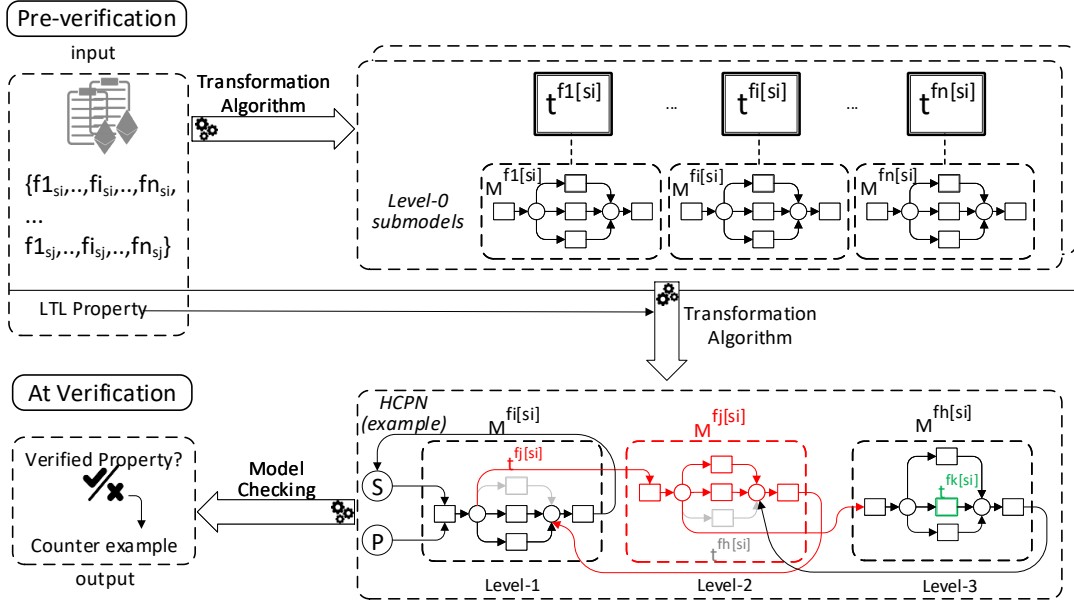


Figure 2: Overview of the approach

model. Then, depending on the property considered for verification, the function targeted in the verification and whether knowledge of the interacting contracts is provided, the actual CPN model that will be passed to the model checker is built by (1) considering the submodel of the function to be checked as the *level-1* of the HCPN, (2) linking it to the places representing the state of the blockchain and (3) building a hierarchy (i.e. additional levels) by explicitly representing function calls in this submodel (if the checked property requires it).

5.1 A CPN Modelling Approach for Solidity Smart Contracts

In our approach, we opt for a hierarchical CPN model to represent a smart contract. As shown in Figure 2, we represent each function of the smart contract by an *aggregated transition* that encapsulates a submodel corresponding to the body of the former. In the first step, these functions are represented disjointedly. In fact, our aim at this pre-verification stage is to get building blocks for the smart contract's model that will be fed to the model checker. In the second step, the obtained submodels are contextualized by specifying the input for the function to be verified (places S and P which respectively represent the state of the contract and the call arguments of the function) and potentially linking its submodel to other functions' submodels in case of the presence of function calls. In fact, function calls are initially abstracted and therefore represented by aggregated transitions in the model (e.g., $t^{fj[si]}$ in Figure 2) under the assumption that they do not present structural problems (deadlock-free and loop-free) which can be separately verified for each function. Depending on the property to be verified, an aggregated transition may need to be *unfolded* if it is involved in the property, hence the multi-level hierarchy in the model (e.g., $t^{fj[si]}$ in $M^{fi[si]}$ is hidden and replaced by its submodel $M^{fj[si]}$).

It is kept *folded* otherwise (e.g., $t^{fk[si]}$ in $M^{fh[si]}$). This abstraction leads to a reduction in the size of the state space the model checker needs to search. We note that this is a generic structure that could be enriched by a more specific control flow if the user provides a source from which a behaviour could be extracted (e.g., the code of an interacting smart contract, a business model defining the workflow in which the contract is used). In the following, we further detail the elements of our proposed model in order to provide a better understanding of the Algorithms 1, 2 and 3.

5.2 Elements of our CPN Model for Solidity

Transitions T

- (1) T^A : aggregated transitions used for the representation of functions, as well as for the modular representation of function calls. They can be substituted by submodels.
- (2) T^R : regular CPN transitions. They are unsubstitutable.

For a transition $t \in T$ we note:

- $t.name$, the name of the transition t
- $t.st$, the Solidity code associated to t
- $t.metaColour$, the metaColour of the control flow places of t (if $t \in T^A$)
- $t.data$, the set of data places associated to t (if $t \in T^A$)
- $t.sub$, the CPN submodel associated to transition t (if $t \in T^A$), with $t.sub.inTransitions$ designating its input (source) transitions and $t.sub.outTransitions$ designating its output (sink) transitions
- $t.guard$, the guard of the transition t
- $t[cf] \in P_{CF} \cup P_S$, input control flow place of t
- $t[input] \in P_p$, input parameters place of t
- $t[data] \subseteq P_{data}$, input data places of t
- $t[cf] \in P_{CF} \cup P_S$, output control flow place of t
- $t[output] \in P_R$, output return place of t

- $t \bullet [data] \subseteq P_{data}$, output data places of t

Places P

- C -flow places P_{CF} , Data places P_{data} , Parameter places P_P , Return places P_R
- a state place $p_s \in P_S$
- a parameters place $p_p \in P_P$

Expressions E An expression is a construct that can be made up of literals, variables, function calls and operators, according to the syntax of Solidity, that evaluates to a single value. For ease of representation later, we define three types of expressions:

- expressions with variables E_V : are expressions that make use of at least one local variable. In such an expression e_v , the set of variables used is accessible via $e_v.vars$.
- expressions with function calls E_F : are expressions that make use of at least one function call. In an expression e_v , the set of function calls used is accessible via $e_v.fctCalls$
- explicit expressions E_E : are expressions that do not make use of any variables nor function calls.

We note that an expression e can of course have both variables and function calls ($e \in E_V \wedge e \in E_F$).

Statements S

- compound statement $\{st[1]; st[2]; \dots; st[N]\}$ (where $\forall i \in [1..N], st[i] \in S$)
- a simple statement (st_{LHS}, st_{RHS}) (where $st_{LHS} \in E$ and $st_{RHS} \in E$)
 - a function call statement, where: $st_{LHS} = \emptyset$ and $st_{RHS}.vars$ designates the set of variables used in the arguments of the call (if $st_{RHS} \in E_V$)
 - an assignment statement, where: $st_{LHS} \in E_V$ and $st_{LHS}.vars$ contains one variable that designates the assigned one $st_{RHS}.vars$ designates the set of variables used in the assignment expression (if $st_{RHS} \in E_V$) and $st_{RHS}.fctCalls$ designates the set of function calls (if $st_{RHS} \in E_F$)
- a control statement⁵

5.3 Transformation Algorithm

The end goal of our transformation is to generate a multi-level hierarchical CPN model that represents the execution of a function of the Solidity smart contract with regard to a property to be verified. First, we generate the aggregated transitions for the smart contract's functions, and build their submodels.

In fact, we see a smart contract function as a set of statements. A statement can be either a compound, a simple or a control one. A simple statement can be a function call, an assignment, a variable declaration, a sending or a returning statement. A control statement can be a requirement, a selection or a loop (a *for* or a *while* loop). To each one of these statement types we define a corresponding pattern in CPN, according to which a snippet of a CPN model is generated [12]. The resulting snippets are linked according to the function's internal workflow. For example, Figure 3 represents the resulting submodel obtained by applying our transformation algorithm (Algorithm 1) on the function *play()* of the *EtherMilestone* contract. The *BUILDCOMPOUNDSTATEMENT* is first called on the body of the function, creating the places $P1$ to $P4$ and then

⁵See footnote 1 for the full list of elements.

Algorithm 1: *CREATESUBMODEL*(t ; st ; p_{in} ; p_{out})

Input : t , statement st , cf input place p_{in} , cf output place p_{out}

Output: submodel of transition t

```

1 switch  $st$  do
2   case compound  $st \{st[1]; st[2]; \dots; st[N]\}$  do
3     BUILDCOMPOUNDSTATEMENT
4     ( $t; st; p_{in}; p_{out}$ )
5   end case
6   case simple  $st$  do
7     switch  $st$  do
8       case ... do
9         ...
10      end case
11      case function call  $st$  do
12        BUILDFUNCTIONCALLSTATEMENT
13        ( $t; st; p_{in}; p_{out}$ )
14      end case
15    end switch
16  end case
17 end switch
```

recursively calling *CREATESUBMODEL* on each of the 5 statements it comprises. The algorithm corresponding to the statement's type is invoked each time, adding the necessary places and transitions in conformance to the defined patterns.

We only include the algorithm responsible for the generation of the CPN pattern for a function call⁶.

Algorithm 2: *BUILDFUNCTIONCALLSTATEMENT*(t ; st ; p_{in} ; p_{out})

Input : transition t , a function call statement
 $st = (st_{LHS}, st_{RHS})$, control flow input place p_{in} ,
control flow output place p_{out}

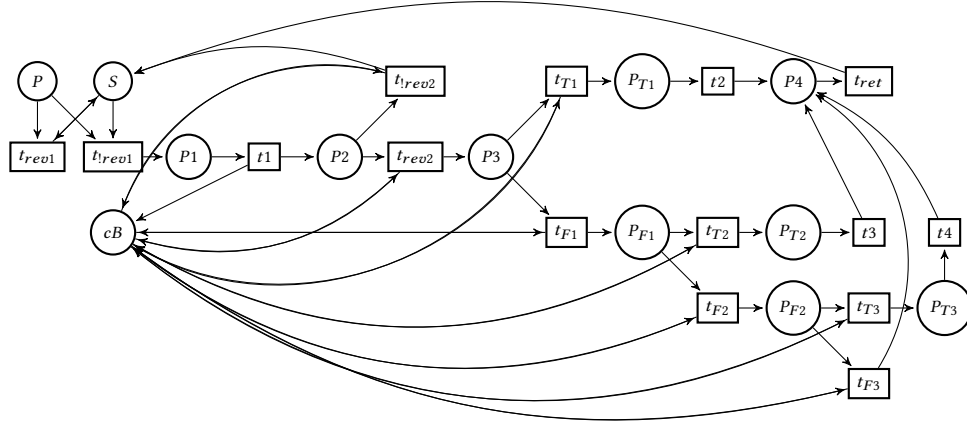
Output: submodel for statement st

```

1 create transition  $t^f$ 
2 create place  $p_{param_f}$ 
3 create arc from  $p_{in}$  to  $t^f$ 
4 create arc from  $p_{param_f}$  to  $t^f$ 
5 CONNECTLOCALVARS( $f_{RHS}.vars; t; t^f$ )
6 CONNECTFUNCALLS( $f_{RHS}.fctCalls; t$ )
7 create arc from  $t^f$  to  $p_{out}$  with a placeholder inscription
```

As stated before, the second step of our modelling approach consists in contextualizing the function to be verified. To do so, two places are created to represent the state of the smart contract and the call arguments for the function in question and are linked to its respective submodel which represents the first level in our hierarchical CPN. Aggregated transitions within this submodel are unfolded (and aggregated transitions within these unfolded transitions, recursively) depending on the property to be verified (see

⁶See footnote 1 for the rest of the algorithms with detailed explanations (in the *Solidity2CPN* document).

Figure 3: CPN submodel of *play()* function in *EtherMilestone***Algorithm 3:** UNFOLDTRANSITION($t^a; p_{in}; p_{out}$)**Input** : aggregated transition t^a , p_{in} , p_{out} **Output**: submodel replacement of t^a

```

1 for  $t' \in t^a.sub.inTransition$  do
2   replicate (arc from  $p_{in}$  to  $t^a$ ) to  $t'$ 
3   replicate (arc from  $\bullet t[input]$  to  $t^a$ ) to  $t'$ 
4   for  $p \in \bullet t^a[data] \cup \bullet t^a[output]$  do
5     replicate (arc from  $p$  to  $t^a$ ) to  $t'$ 
6   end for
7 end for
8 for  $t' \in t^a.sub.outTransition$  do
9   replicate (arc from  $t^a$  to  $p_{out}$ ) to  $t'$  with the placeholder
   inscription replaced by values from  $\bullet t'[cf]$ 
10 end for
11 hide transition  $t^a$  and all arcs linked to it

```

algorithm UNFOLDTRANSITION) which builds the final hierarchical CPN to be checked.

6 FORMAL VERIFICATION OF SMART CONTRACTS

As explained in Section 5, we adopt a two-phase verification approach, in which we rely on *Helena* to verify LTL properties that express the susceptibility of contracts to vulnerabilities. To this aim, we start by expressing each targeted vulnerability in LTL.

6.1 Expressing Vulnerabilities in LTL

In the following, $M_{s_i}^f$ designates the CPN submodel corresponding to function f in smart contract s_i . We note that sometimes we use parameterized propositions to indicate that they are applied to an unspecified aggregated transition. Concretely, such propositions need to be explicitly defined for each transition to be verified⁷.

⁷Not to be confused with first order predicates

6.1.1 Integer Overflow/Underflow. In our CPN model, we define correspondences between the types used in the Solidity language and those offered by *helena* so that they cover the same ranges. The model checker is therefore able to detect when the smart contract contains an out-of-range expression. It does not, however, pinpoint the source of the anomaly, so the user does not have much information to go on to track it and try to correct it. To overcome this deficiency, we propose to model integer overflows/underflows as a safety LTL property that can be verified on a specific variable x :

$$IUO_x = \Box \neg xIsOutOfRange$$

Where $xIsOutOfRange$ is a proposition that evaluates to true if the value of x is not included in the range of its type which we delimit by defining lower and higher thresholds (*minThreshold* and *maxThreshold* respectively).

$$xIsOutOfRange = (x < minThreshold) \vee (x > maxThreshold)$$

6.1.2 Reentrancy. This vulnerability is related to functions that contain instructions responsible for Ether transfer, and therefore is applied w.r.t a function containing a *sending* statement. Given such a function, we propose two LTL properties. The first is a safety property defined as follows:

$$Reentrancy_{M_{s_i}^f} = \Box \neg reentrant_{M_{s_i}^f}$$

Where $reentrant_{M_{s_i}^f}$ is true if the necessary condition under which a reentrancy vulnerability can be detected in the function f in the smart contract s_i is valid. This condition can only be defined when the user indicates the variable x serving as a record for balances and whose assignment should be watched. Such a condition expresses the presence of a *sending* statement which is not preceded by an assignment to x :

$$reentrant_{M_{s_i}^f} = (\neg XAssignment) \mathcal{U} Sending$$

Where $XAssignment$ is true when a statement is an assignment to the variable x and *Sending* is true when a statement is a sending one.

A vulnerability is detected when $Reentrancy(t_{s_i}^f)$ evaluates to false. This property is used when we only have the code of the smart contract to be verified (i.e., a totally free behaviour). If the code of the interacting contract s_j is available, we propose the following LTL property:

$$Reentrancy_{M_{s_i}^f} = SendingTo_{s_j} \rightarrow \bigcirc \square ((\neg SendingTo_{s_j}) \mathcal{U} endOfFallback_{s_j})$$

Using this property we can verify that once the sending statement is executed ($SendingTo_{s_j}$ is true), it cannot be executed again until the fallback function of the receiving contract has finished ($endOfFallback_{s_j}$ is true) i.e., no reentrancy breach can happen.

6.1.3 Self-destruction. It is checked by detecting the presence of a test containing *this.balance* in the code of the function:

$$selfDestruction_{M_{s_i}^f} = \neg testOnBalance_{M_{s_i}^f}$$

This detection process can be further enhanced when the code of the interacting smart contract is available. In that case, given a function g in s_j that contains a self destruction instruction directing Ether to s_i , a function f in s_i is considered safe against this vulnerability if it does not contain a test on *this.balance* or if g cannot be executed prior to f under inspection, which is expressed by the LTL property:

$$selfDestruction_{M_{s_i}^f} = (\neg testOnBalance_{M_{s_i}^f}) \vee (\neg selfDestruct_{M_{s_j}^g} \mathcal{U} start_{M_{s_i}^f})$$

We note that even though these properties can detect the presence of the self destruction vulnerability, more information on what the function exactly does needs to be provided in order to be able to assess its harmfulness on the execution. This can still be checked by evaluating a contract-specific property.

6.1.4 Timestamp Dependence. In order to check for this vulnerability, we propose an LTL property to detect the accessibility of *block.timestamp* or its alias *now*:

$$TSD_{M_{s_i}^f} = \square \neg TimestampDependantStatement$$

Where *TimestampDependantStatement* is true if a *timestamp* is used in a statement. Similarly to the self destruction vulnerability, the presence of timestamp dependence can be detected using the proposed property, but to check the harm it may incur a more appropriate contract-specific property needs to be evaluated.

6.1.5 Skip Empty String Literal. This can be checked for a function f containing function calls by verifying that for any function call with n arguments $\{a_1, \dots, a_n\}$ no empty string is passed as an argument (except for the last one a_n). We express this as follows:

$$SkipEmpty_{M_{s_i}^f} = \square \neg FunctionCall$$

Where *FunctionCall* is true when the statement is a function call with an empty argument a_i ($i \neq n$).

6.1.6 Uninitialized Storage Variable. This is verified for a function f where a variable x of a complex type is defined, by checking the following property:

$$UninitializedVariable_{M_{s_i}^f} = \neg readX \mathcal{U} writeX$$

Where *readX* is true when x is read in a statement and *writeX* is true when it is assigned.

6.2 Model Checking of Smart Contracts

Once we have applied our transformation algorithm to first generate the submodels of a smart contract's functions, verifying properties of the contract would come down to verifying properties on the corresponding CPN model. For model checking, we chose *Helena* [8] which is a High LEvel Nets Analyzer available as a command line tool. It offers explicit model checking support for an on-the-fly verification of state and LTL properties over CPN models described in *Helena's* specification language.

```
440 proposition outOfRange : exists (t in S | (t->1).gameNum > maxThreshold
or (t->1).gameNum < minThreshold);
1 ltl property IUO:
2 [] not outOfRange;
```

Figure 4: Integer overflow/underflow property in *Helena*

We have generated the CPN models of our use cases for *Helena* using our prototype for the transformation algorithm (see Figure 3 for an example of a visual representation of the CPN submodel of the function *play()* in *EtherMilestone*). We have then written the properties for the vulnerabilities in Section 4 in *Helena's* language. Thus, we were able to detect the described vulnerabilities, as well as contract-specific properties established for our examples. The artifacts used in this verification, a detailed report on the results and the prototype implementation are available in our repository⁸.

```
Search report
-----
Action performed
  property checking
Host machine
  Ikramz (pid = 60511)
Property checked
  IUO
Termination state
  PROPERTY_VIOLATED
Statistics report
-----
Model statistics
-----
  24 places
  28 transitions
  72 arcs
Trace report
-----
The following run invalidates the property.
{
  S = <({0, 0, 0, 1}, false, 0) >
  P_RestartLotto = <({0, 0}, 0) >
  P_PlayTicket = <({1, 10}, 10, 1) > + <({2, 10}, 10, 2) > + <({3, 1
```

Figure 5: Model checking result

Figure 4 shows the corresponding property written in *Helena* for the *IUO* LTL property applied on the variable *gameNum* in *EtherLotto* and Figure 5 is a snippet of the result of the model checker showing the detection of the vulnerability and the indication of a counter example. The detection process took 35.8s including 1.79s for source compilation and 34.01s for state space exploration which had entailed the processing of 13 664 828 states, for a test on a model with 10 players.

⁸See footnote 1

7 CONCLUSION AND FUTURE WORK

Considering the crucial role the blockchain technology plays in many domain applications especially with its increasingly expanding reach, it is critical for smart contracts to provide certain guarantees in terms of correctness to support a foundation built on trust. Formal approaches for the verification of Solidity smart contracts have been proposed, but they are generally designed to target specific vulnerabilities known in the literature (e.g., reentrancy) which have been reported to be the root of some attacks or malfunctions. Checking the absence of such vulnerabilities in a smart contract, however necessary, does not guarantee its correctness as a faulty behaviour may stem from a flaw specific to that contract. With our proposed approach we aim to bring a solution to this problem by providing a way to formally verify smart contracts by not only checking for vulnerabilities in the code but also offering the possibility to express additional contract-specific properties to check. Our prototype and preliminary tests prove the feasibility of our approach. A comprehensive evaluation (e.g., by experimenting on some existing data set of smart contracts like *SolidiFI*⁹) still needs to be carried out. To further improve our verification results and get better scalability, we intend to work on *Helena* by embedding it with an extension to an existing state space reduction technique previously developed for PNs [17] and adapting it for CPNs.

REFERENCES

- [1] [n. d.]. Solidity documentation. <https://docs.soliditylang.org/en/latest/>.
- [2] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 66–77.
- [3] Saswat Anand, Corina S. Pasareanu, and Willem Visser. 2009. Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.* 11, 1 (2009), 53–67.
- [4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. [n. d.]. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*.
- [5] 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. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Austria, October 24*.
- [6] Ting Chen, Xiaoli Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 442–446.
- [7] Wang Duo, Xin Huang, and Xiaofeng Ma. 2020. Formal Analysis of Smart Contract Based on Colored Petri Nets. *IEEE Intell. Syst.* 35, 3 (2020), 19–30.
- [8] Sami Evangelista. 2005. High Level Petri Nets Analysis with Helena. In *Applications and Theory of Petri Nets 2005*. Berlin, 455–464.
- [9] Ikram Garfatta, Kais Klai, Walid Gaaloul, and Mohamed Graïet. 2021. A Survey on Formal Verification for Solidity Smart Contracts. In *ACSW '21: 2021 Australasian Computer Science Week Multiconference, Dunedin, New Zealand, 1-5 February, 2021*. ACM, 3:1–3:10.
- [10] Ikram Garfatta, Kais Klai, Mohamed Graïet, and Walid Gaaloul. 2018. Formal Modelling and Verification of Cloud Resource Allocation in Business Processes. In *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 11229. Springer, 552–567.
- [11] Ikram Garfatta, Kais Klai, Mohamed Graïet, and Walid Gaaloul. 2020. Blockchain-Based Business Processes: A Solidity-to-CPN Formal Verification Approach. In *Service-Oriented Computing - ICSOC 2020 Workshops - PhD symposium, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings (Lecture Notes in Computer Science)*, Vol. 12632. Springer, 47–53.
- [12] Ikram Garfatta, Kais Klai, Mohamed Graïet, and Walid Gaaloul. 2021. A Solidity-to-CPN Approach Towards Formal Verification of Smart Contracts. In *30th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2021, France, October 27-29, 2021*. IEEE, to appear.
- [13] Kurt Jensen. 1989. Coloured Petri nets: A high level language for system design and analysis. In *International Conference on Application and Theory of Petri Nets*. 342–416.
- [14] Kurt Jensen and Lars M. Kristensen. 2009. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated.
- [15] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS*.
- [16] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Poland, April 7-11, Proceedings*.
- [17] Kais Klai and Denis Poitrenaud. 2008. MC-SOG: An LTL Model Checker Based on Symbolic Observation Graphs. In *Applications and Theory of Petri Nets, 29th International Conference, PETRI NETS 2008, Xi'an, China, June 23-27, 2008. Proceedings*. 288–306.
- [18] Zhen-Tian Liu and Jing Liu. 2019. Formal Verification of Blockchain Smart Contract Based on Colored Petri Net Models. In *43rd IEEE COMPSAC*. 555–560.
- [19] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 254–269.
- [20] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, St. Kitts and Nevis, February 18-22, 2019*. 446–465.
- [21] Robin Milner, Mads Tofte, and Robert Harper. 1990. *Definition of standard ML*. MIT Press.
- [22] T. Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.
- [23] Elthon A. S. Oliveira, Hyggo Oliveira de Almeida, and Leandro Dias da Silva. 2007. Formal modelling and verification of a component model using coloured petri nets and model checking. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15, 2007*. ACM, 1427–1431.
- [24] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence*. IEEE Computer Society, 46–57.
- [25] Kristin Y. Rozier. 2011. Linear Temporal Logic Symbolic Model Checking. *Comput. Sci. Rev.* 5, 2 (2011), 163–203.
- [26] M. Staples, S. Chen, Sara Falamaki, A. Ponomarev, Paul Rimba, A. Tran, I. Weber, Sherry Xu, and John Zhu. 2017. Risks and opportunities for systems using blockchain and smart contracts. In *Data61 (CSIRO), Sydney*.
- [27] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th ACSAC*. 664–676.
- [28] KM Van Hee and PAC Verkoulen. 1991. Integration of a data model and high-level petri nets. In *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*.

⁹<https://github.com/smartbugs/SolidiFI-benchmark>