Towards Automated Verification of Smart Contract Fairness

Ye Liu
li0003ye@ntu.edu.sg
Nanyang Technological University
Singapore

Yi Li
yi_li@ntu.edu.sg
Nanyang Technological University
Singapore

Shang-Wei Lin
shang-wei.lin@ntu.edu.sg
Nanyang Technological University
Singapore

Rong Zhao
rong.zhao@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Smart contracts are computer programs allowing users to define and execute transactions automatically on top of the blockchain platform. Many of such smart contracts can be viewed as games. A game-like contract accepts inputs from multiple participants, and upon ending, automatically derives an outcome while distributing assets according to some predefined rules. Without clear understanding of the game rules, participants may suffer from fraudulent advertisements and financial losses. In this paper, we present a framework to perform (semi-)automated verification of smart contract fairness, whose results can be used to refute false claims with concrete examples or certify contract implementations with respect to desired fairness properties. We implement FairCon, which is able to check fairness properties including truthfulness, efficiency, optimality, and collusion-freeness for Ethereum smart contracts. We evaluate FairCon on a set of real-world benchmarks and the experiment result indicates that FairCon is effective in detecting property violations and able to prove fairness for common types of contracts.

CCS CONCEPTS

• **Software and its engineering → Software verification**; **Software verification and validation**.

KEYWORDS

Smart contract, fairness, mechanism design.

ACM Reference Format:
Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards Automated Verification of Smart Contract Fairness. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409740

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.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409740

1 INTRODUCTION

The blockchain technology has been developed rapidly in recent years, since the introduction of Bitcoin [42] by Nakamoto in 2008. The distributed and tamper-resistant nature of blockchain has made it the perfect platform for hosting smart contracts. Smart contracts are computer programs running atop blockchain platforms to manage large sums of money, carry out transactions of assets, and govern the transfer of digital rights between multiple parties. Ethereum [60] and EOS [28] are among the most popular blockchain platforms which support smart contracts and have them applied in many areas. As of February 2020, there are over one million smart contracts deployed on Ethereum, which is a 100 fold increase since just two years ago. These smart contracts have enabled about 2.7K decentralized applications (DApps) [3] serving 20K daily users on finance, health, governance, gambling, games, etc.

The security of smart contracts has been at the center of attention, ever since their adoption in the management of massive monetary transactions. One of the most notorious cases is the DAO attack [49] on Ethereum, which resulted in a loss of $60 million worth, due to the *reentrancy vulnerability* being exploited by malicious attackers. Several gambling games on EOS, including `EOS.WIN` and `EOSPlay`, were recently hacked using a technique called the *transaction congestion attack* [47] and led to significant asset loss. What these incidents share in common is that certain security vulnerabilities neglected during contract development are exploited by malicious parties, causing a loss for the contract owners (and possibly other honest participants). These vulnerabilities are programming errors, indicating a mismatch between the contract developers' expectations and how the contract code actually works. They are easy to detect once the vulnerability patterns are recognized. In fact, much research has been dedicated to preventing, discovering, and mitigating such attacks.

In contrast, the *fairness* issues of smart contracts have not yet attracted much attention. A smart contract is unfair to certain participants if there is a mismatch between the participants' expectations and the actual implementation of the game rules. It is possible that a malicious party may gain an advantage over others through the exploitation of security vulnerabilities, e.g., examining other participants' actions in a sealed game. In this paper, we would like to focus more on the fairness issues introduced by the logical design of the contracts instead, which are orthogonal to the security issues. For example, smart contracts may well be advertised as "social games" with a promised 20% return for any investment, but turn out to be "Ponzi schemes" [11]. In this case, the possibility that the game may

eventually slow down and never pay back is intentionally left out. Similarly, many auction DApps claim to be safe and fair, yet it is still possible for bidders to collude among themselves or with the auctioneer to make a profit at the expenses of the others [61]. The fairness issues mostly reside in contract logic: some of them are unfair by design, while the rest are careless mistakes. This makes the detection of such issues particularly challenging, because every case can be different and there is no hope in identifying predefined patterns. Since it is often not the contract creators' interest at risk (or even worse when they gain at the expenses of participants), there is little incentive for them to allocate resources in ensuring the fairness of their contracts. On the other hand, it is rather difficult, for inexperienced users, to tell whether a contract works as advertised, even with the source code available.

In this paper, we present FairCon, a framework for verifying fairness properties of smart contracts. Since general fairness is largely a subjective concept determined by personal preferences, there is no universal truth when considering only a single participant. We view a smart contract as a game (or mechanism [29, 45]), which accepts inputs from multiple participants, and after a period of time decides the outcome according to some predefined rules. Upon game ending, each participant receives certain utility depending on the game outcome. With such a mechanism model, we can then verify a wide range of well-studied fairness properties, including *truthfulness*, *efficiency*, *optimality*, and *collusion-freeness*. It is also possible to define customized properties based on specific needs.

The real challenge in building the fairness verification framework is on how to translate arbitrary smart contract code into standard mechanism models. Our solution to this is to have an intermediate representation for each type of games, which has direct semantic translation to the underlying mechanism model. For instance, the key components in an auction are defined by the set of bidders, their bids, and the allocation and clear price rules of the goods in sale. To synthesize the intermediate mechanism model for an auction smart contract, we first manually instrument the contract code with customized labels highlighting the relevant components. Then we perform automated *symbolic execution* [35] on the instrumented contract to obtain symbolic representations for auction outcomes in terms of the actions from a bounded number of bidders. This is finally mapped to standard mechanism models where fairness properties can be checked. We either find property violations with concrete counterexamples or are able to show satisfaction within the bounded model. For properties of which we do not find violation, we attempt to prove them for unbounded number of participants on the original contract code, with program invariants observed from the bounded cases.

By introducing the intermediate representations, we could keep the underlying mechanism model and property checking engine stable. We defined an intermediate language for two types of game-like contracts popular on Ethereum, i.e., auction and voting. We implemented FairCon to work on Ethereum smart contracts and applied it on 17 real auction and voting contracts from `Etherscan` [5]. The effort of manual labeling is reasonably low, considering the structural similarity of such contracts. The experimental results show that there are many smart contracts violating fairness property and FairCon is effective to verify fairness property and meanwhile achieves relatively high efficiency.

Contributions. Our main contributions are summarized as follows.

- We proposed a general fairness verification framework, FairCon, to check fairness properties of smart contracts. In particular, we demonstrated FairCon on two types of contracts and four types of fairness properties.
- We defined intermediate representations for auction and voting contracts, and designed a (semi-)automated approach to translate contract source code into mathematical mechanism models which enable fairness property checking.
- In addition to discovering property violations for bounded models, we apply formal verification to prove satisfaction of properties for the unbounded cases as well.
- We implemented FairCon and evaluated it on 17 real-world Ethereum smart contracts. The results show that FairCon is able to effectively detect fairness violations and prove fairness properties for common types of game-like contracts. The dataset, raw results, and prototype used are available online: https://doi.org/10.21979/N9/0BEVRT.

Organizations. The rest of the paper is organized as follows. Section 2 illustrates the workflow of FairCon with an example. Section 3 presents a general mechanism analysis model and defines a modeling language customized for auction and voting contracts, serving as an intermediate representation between the contract source code and the underlying mechanism model. We then describe the model construction and fairness checking as well as verification techniques in Sect. 4. Section 5 gives details on the implementation and presents the evaluation results. Sections 6 and 7 compare FairCon with the related work and conclude the paper, respectively.

2 FAIRCON BY EXAMPLE

In this section, we use an auction contract to illustrate how our approach works in constructing the intermediate mechanism model and verifying fairness properties.

Example 2.1. Figure 1 shows a simplified Ethereum smart contract, named `CryptoRomeAuction`, written in Solidity [53], taken from `Etherscan`.[1] The contract implements a variant of open English auction for a blockchain-based strategy game, where players are allowed to buy virtual lands with cryptocurrencies. The auction is given a predefined life cycle parameterized by start and end times. A participant can place a bid by sending a message to this contract indicating the value of the bid. The address of the participant and the bid amount are stored in variables `msg.sender` and `msg.value`, respectively. The address of the current highest bidder is recorded in `highestBidder` (Line 9), and a mapping `refunds` is used to keep the contributions of each participant (Line 10) for possible refunding later. The `bid()` function (Lines 11–21) is triggered upon receiving the message. The bid is rejected if the bid amount is no more than the sum of the current highest bid and the minimal increment value `duration` (Lines 13–15). Otherwise, the previous `highestBidder` gets a refund (Lines 16–18), and the `highestBidder` (Line 19) and `highestBid` (Line 20) are updated accordingly.

[1] https://etherscan.io/address/0x760898e1e75dd7752db30bafa92d5f7d9e329a81

```
1  contract CryptoRomeAuction {
2    /** FairCon Annocations
3     @individual(msg.sender, msg.value, VALUE)
4     @allocate(highestBidder)
5     @price(highestBid)
6     @outcome(bid())
7    */
8    uint256 public highestBid = 0;
9    address payable public highestBidder;
10   mapping(address=>uint) refunds;
11   function bid() public payable{
12     uint duration = 1;
13     if (msg.value < (highestBid + duration)){
14       revert();
15     }
16     if (highestBid != 0) {
17       refunds[highestBidder] += highestBid;
18     }
19     highestBidder = msg.sender;
20     highestBid = msg.value;
21   }
22 }
```

Figure 1: The CryptoRomeAuction Solidity source code.

Table 1: Example instances of CryptoRomeAuction.

	Truthful			Untruthful			Collusion		
Bidder	p_1	p_2	p_3	p_1	p_2	p_3	p_1	p_2	p_3
Valuation	3	4	6	3	4	6	3	4	6
Bid	3	4	6	3	4	5	3	0	4
Allocation	✗	✗	✓	✗	✗	✓	✗	✗	✓
Price	0	0	6	0	0	5	0	0	4
Utility	0	0	0	0	0	1	0	1	1

Threats to Contract Fairness. One way that `CryptoRomeAuction` can become unfair to the participants is through the so called *shill bidding* [30]—a shill tries to escalate the price without any intention of buying the item. This can be induced by either the auctioneer or adversarial participants, and other bidders may need to pay more as a result. Occasionally, the shill wins the auction if no other higher bid comes before auction ends. The item may then be sold again at a later time.

Apart from shill bidding, there are a number of other well-studied properties from the game theory and mechanism design literature, which can be used to evaluate the fairness of an auction. We use the example instances shown in Table 1 to demonstrate. Suppose there are three bidders, p_1, p_2, and p_3, participating in the auction. Each of them has a valuation of the item, i.e., the item worth three, four, and six units of utility for p_1, p_2, and p_3, respectively. The Columns "Truthful", "Untruthful", and "Collusion" in Table 1 show the three example scenarios, where the players act truthfully, untruthfully, and collude among themselves. The Rows "Bid", "Allocation", "Price", and "Utility" show the bids placed, the final allocation of the item, the clear price, and the utilities obtained by the bidders, respectively.

Same as other first-price auction schemes, `CryptoRomeAuction` is not *truthful*, i.e., bidding truthfully according to one's own valuation of the item is not a dominant strategy. In the ideal truthful scenario, all bidders bid according to their valuations, and p_3 wins the bid with a utility of zero, because the payment equals to his/her valuation of the item. In another scenario, where p_3 bids five (untruthfully), his/her utility would increase by one because of the lower clear price. This is called *bid shading*, which only affects the revenue from the auction in this example, but may affect other participants' utilities in some other cases.

In the third scenario, p_2 and p_3 collude in order to gain extra profits. With full knowledge of each other's valuations, p_2 and p_3 may decide to form a cartel and perform bid shading. One possibility is to have p_2 forfeit his/her chance and p_3 bids four, and they divide the profit equally among themselves. Each of them gains one unit of utility as a result.

$$
\begin{aligned}
&\texttt{CryptoRomeAuction} := (msgsender_1, msgvalue_1, \_)\\
&\quad (msgsender_2, msgvalue_2, \_)\\
&\quad (msgsender_3, msgvalue_3, \_)\\
&\quad \textbf{assume}: (\textbf{not } (msgvalue_2 < msgvalue_1 + 1)) \textbf{ and}\\
&\qquad (\textbf{not } (msgvalue_3 < msgvalue_2 + 1))\\
&\quad \textbf{allocate}: \textbf{argmax}(msgvalue_1, msgvalue_2, msgvalue_3)\\
&\quad \textbf{price}: \textbf{max}(msgvalue_1, msgvalue_2, msgvalue_3)
\end{aligned}
$$

Figure 2: The mechanism model of CryptoRomeAuction with three bidders.

Checking Fairness Properties. Given a mechanism model abstracting the auction settings, the set of fairness properties are well-defined and can be formally specified based on the model. The main challenge remains on how to extract the underlying mechanism model from the smart contract source code. Now we illustrate how this is done for `CryptoRomeAuction` in FairCon and outline the process of automated property checking as well as verification.

Albeit variations in implementations, all auction contracts share some common components, such as the bidders' identifiers, their bids, and the allocation as well as clear price rules. We rely on users to provide annotations for these components directly on the source code, which are demonstrated on Lines 2–7 in Fig. 1. Specifically, the annotations specify the bidders' information as a tuple, "`@individual(msg.sender,msg.value)`", indicating the variables used to store the identifier and the bid value, respectively. Similarly, "`@allocation(highestBidder)`" and "`@price(highestBid)`" indicate that the allocation result and the clear price are stored in `highestBidder` and `highestBid`, respectively. Finally, "`@outcome`" is used to label the function defining the auction allocation logic.

With these labels, we perform symbolic execution [35] on the `bid()` function treating the participants' inputs—`msg.value`—as *symbolic variables*. The result of this would be two symbolic expressions for both `highestBidder` and `highestBid`, which symbolically represent the allocation and clear price functions, respectively. We can then use these information to synthesize an intermediate mechanism model, shown in Fig. 2. The model is specified in a customized language designed for auction and voting contracts. Details of the language syntax and semantics can be found in Sect. 3. At the high level, the model specifies information of the participating individuals and the auction rules: we consider a bounded model with only three bidders (i.e., $msgsender_1$, $msgsender_2$, and $msgsender_3$), their bids have to satisfy the constraint specified in the `assume` clause, the allocation function is given as "arg max($msgvalue_1$,

$msgvalue_2, msgvalue_3$)", and the clear price function is given as "max($msgvalue_1, msgvalue_2, msgvalue_3$)".

The intermediate mechanism model in Fig. 2 has well-defined mathematical semantics, which can be used to check the desired fairness properties. We encode both the model and the property with an SMT formula such that a counterexample exists if and only if the formula is satisfiable. More details on the encoding can be found in Sect. 4.2. If the formula is unsatisfiable, we are confident that the property holds for the bounded case with three bidders. We then attempt to prove the property by instrumenting the contract program with *program invariants* encoding the allocation and clear price clauses synthesized previously, but parameterized by an unbounded number of bidders. The instrumented program and the property are then passed to a program verification tool, such as Dafny [38], to perform the automated verification.

3 THE ANALYSIS FRAMEWORK FOR SMART CONTRACT FAIRNESS

In this section, we first provide necessary background and definitions on mechanism models and fairness properties well studied in the mechanism design literature [29, 45]. Then we give the abstract syntax and semantics of our mechanism modeling language to support automated model construction and property checking.

3.1 Smart Contracts as Mechanism Models

Mechanism design is used to design economic mechanisms or incentives to help attain the goals of different stakeholders who participate in the designated activity. The goals are mainly related to the outcome that could be described by participants' payoff and their return in the activity. We model the logic behind smart contracts with a mathematical object known as the mechanism.

In a *mechanism model*, we have a finite number of individuals, denoted by $N = \{1, 2, \ldots, n\}$. Each individual i holds a piece of private information represented by a *type*, denoted $\theta_i \in \Theta_i$. Let the types of all individuals be $\theta = (\theta_1, \ldots, \theta_n)$, and the space be $\Theta = \times_i \Theta_i$. The individuals report, possibly dishonestly, a *type (strategy) profile* $\hat{\theta} \in \Theta$. Based on everyone's report, the mechanism model decides an outcome which is specified by an *allocation function* $d : \Theta \mapsto O$, and a *transfer function* $t : \Theta \mapsto \mathbb{R}^n$, where $O = \{o_i \in \{0,1\}^n \mid \Sigma_i o_i = 1\}$ is the set of possible outcomes.

The preferences of an individual over the outcomes are represented using a *valuation function* $v_i : O \times \Theta_i \mapsto \mathbb{R}$. Thus, $v_i(o, \theta_i)$ denotes the benefit that individual i of type θ_i receives from an outcome $o \in O$, and $v_i(o, \theta_i) > v_i(o', \theta_i)$ indicates that individual i prefers o to o'. The individual i's utility under strategy profile $\hat{\theta}$ is calculated by subtracting the payment to be made from the valuation of a certain outcome: $u_i(\hat{\theta}) = v_i(\hat{\theta}, \theta_i) - t_i(\hat{\theta})$.

3.2 Fairness Properties

The fairness of smart contracts is usually subject to the understandings and preferences of the participating parties—a contract fair to someone may be unfair to the others. In particular, fairness can be considered from both the participants' and the contract creators' points of view. To capture such nuances, individual parties have to be modeled separately before such subjective fairness properties can be specified against the model.

Generally speaking, all properties which can be expressed in terms of the mechanism model defined in Sect. 3.1 are supported by our reasoning framework. To keep the presentation simple, in this paper, we focus on analyzing a set of generic fairness properties based on the mechanism models. We restrict the discussion to four types of well-studied properties in the literature, namely, truthfulness, optimality, efficiency, and collusion-freeness.

To formally define the properties, we first introduce an important concept—*dominant strategy*. We use $\hat{\theta}_{-i}$ to denote the strategy profile of the individuals other than i, i.e., $(\hat{\theta}_1, \ldots, \hat{\theta}_{i-1}, \hat{\theta}_{i+1}, \ldots, \hat{\theta}_n)$. Therefore, $(\hat{\theta}'_i, \hat{\theta}_{-i})$ is used to denote the strategy profile which differs from $\hat{\theta}$ only on $\hat{\theta}_i$.

Definition 3.1 (Dominant Strategy). A strategy $\hat{\theta}_i \in \Theta_i$ is a dominant strategy for i, if $\forall \hat{\theta}_{-i} \forall \hat{\theta}'_i \in \Theta_i \cdot u_i(\hat{\theta}_i, \hat{\theta}_{-i}) \geq u_i(\hat{\theta}'_i, \hat{\theta}_{-i})$. When equality holds, the strategy is a weak dominant strategy.

We say that a mechanism model is truthful if and only if for any individual and strategy profile, reporting one's real type (truth-telling, i.e., $\forall i \in N \cdot \hat{\theta}_i = \theta_i$) is a dominant strategy.

Definition 3.2 (Truthfulness). Formally, a mechanism is truthful if and only if, $\forall \theta_{-i} \forall \hat{\theta}_i \in \Theta_i \cdot u_i(\theta_i, \theta_{-i}) \geq u_i(\hat{\theta}_i, \theta_{-i})$.

Given an auction smart contract with many bidders competing for a single indivisible good, the account which creates the contract is the *auctioneer* and the accounts which join the auction are the *bidders*. If the auction prevents bidders from benefiting more by bidding less, it is truthful. When bidding untruthfully is not a good strategy, the auction can generally attract more honest bidders and the auctioneer can get higher revenue for the good on sale.

Definition 3.3 (Efficiency). We say a mechanism is efficient if and only if its allocation function achieves maximum total value, i.e., $\forall \hat{\theta} \in \Theta \forall d' \cdot \sum_i v_i(d(\hat{\theta}), \theta_i) \geq \sum_i v_i(d'(\hat{\theta}), \theta_i)$.

Suppose no bidder can affect any other bidder's valuation. If the only winner is the bidder who values the good the most, the auction is efficient.

Definition 3.4 (Optimality). We say a mechanism is optimal if and only if its transfer function achieves maximum total net profit, i.e., $\forall \hat{\theta} \in \Theta \forall t' \cdot \sum_i t_i(\hat{\theta}) \geq \sum_i t'_i(\hat{\theta})$.

Similarly, if the winner is the one who bids the highest, the auction is optimal. In this case, the auctioneer receives the highest revenue.

We use $\hat{\theta}_{-ij}$ to denote the strategy profile of individuals other than i and j, i.e., $\{\hat{\theta}_1, \ldots, \hat{\theta}_{i-1}, \hat{\theta}_{i+1}, \ldots, \hat{\theta}_{j-1}, \hat{\theta}_{j+1}, \ldots, \hat{\theta}_n\}$.

Definition 3.5 (2-Collusion Free). We say a mechanism is 2-collusion free if there does not exist a cartel of individuals i and j, whose untruthful strategies increase the group utility, formally, $u_i(\hat{\theta}_i, \hat{\theta}_j, \theta_{-ij}) + u_j(\hat{\theta}_i, \hat{\theta}_j, \theta_{-ij}) \geq u_i(\theta_i, \theta_j, \theta_{-ij}) + u_j(\theta_i, \theta_j, \theta_{-ij})$.

Collusion is a big concern in auction and other multi-player games. The basic 2-collusion freeness property in an auction means that any two bidders' collusion cannot help them achieve higher gain. This prevents bid price manipulation to a certain extent, which helps guarantee fair chance for all bidders and maintain good revenue for the auctioneer. A more general version, i.e., n-collusion freeness, can be defined in a similar way.

```
<individual> := (id : 𝕊, bid : ℕ, val : ℕ)
       <func> := max | argmax
        <exp> := <individual>.id | <individual>.bid
               | ℕ | <exp> [+-] <exp> | <func>(<exp>*)
       <bool> := <exp> == <exp> | <exp> < <exp>
               | not <bool> | <bool> and <bool>
<assumption> := assume : <bool>
    <outcome> := allocate : <exp>
               | price : <exp>; allocate : <exp>
   <property> := <bool> | forall : <bool>
  <mechanism> := <individual>*; <assumption>; <outcome>;
                 <property>
```

Figure 3: Syntax of the auction/voting mechanism model.

$$\frac{(id_1, bid_1, val_1), \ldots, (id_n, bid_n, val_n)}{N \leftarrow \{1, \ldots, n\} \quad \hat{\theta} \leftarrow \{bid_1, \ldots, bid_n\} \quad \{v_i(o_i, \theta_i)\} \leftarrow \{val_i\}} \text{ [Indiv]}$$

$$\frac{\textbf{assume} : assumption \qquad \textbf{allocate} : allocation}{d(\hat{\theta}) \leftarrow \textbf{eval}(assumption \wedge allocation)} \text{ [Alloc]}$$

$$\frac{\textbf{assume} : assumption \qquad \textbf{price} : clearprice}{t(\hat{\theta}) \leftarrow \textbf{eval}(assumption \wedge clearprice)} \text{ [Price]}$$

Figure 4: Semantic rules of the auction/voting model.

3.3 Mechanism Modeling Language

We now propose a domain-specific language to facilitate the automated translation from smart contracts to mechanism models.

We define an abstract syntax of the mechanism modeling language, which is applicable to both auction and voting. Figure 3 shows the context-free grammar of the language. A mechanism model comprises one or more *individuals*, an *assumption*, an *outcome*, and a *property* to be verified. An individual is defined as a triple containing the identifier "*id*", bid amount "*bid*", and valuation "*val*". An assumption is a Boolean constraint which should be satisfied upon the entry of the contract. The outcome of the contract is specified by the allocation and the clear price functions, which are expressions over *id* and *bid*. Voting contract typically does not have a clear price function. We allow properties to be specified using a Boolean expression optionally preceded by a "forall" quantifier.

Language Semantics. The semantic mapping from the modeling language to the underlying mechanism model is summarized in Fig. 4. The "Indiv" rule maps the individuals and their reported types as well as valuations. More specifically, the individuals' bids are mapped to their reported types $\hat{\theta}$, and an individual of type θ_i's valuation of the item $v_i(o_i, \theta_i)$ is val_i, where o_i denoted the outcome where the item is allocated to i. The "Alloc" rule conjuncts the Boolean expression *assumption* from the "assume" clause and the symbolic expression *allocation* in terms of individuals' strategies from the "allocate" clause, which is evaluated as the allocation function. Similarly, the transfer function is the conjunction of the *assumption* and the *clearprice* expressions. There are some differences between auction and voting: clear price is absent from voting, where allocation is done by comparing the number of ballots (bids) by the participating individuals; whereas in auction, the individuals who bid no less than the clear price can be allocated the item.

This language works for the most commonly seen auction and voting contracts with fairness concerns. For example, Ethereum smart contracts meeting the ERC-1202 (voting) [4] and ERC-1815 (blind auctions, under review) [6] standards all follow the same interface and structure, therefore can be automatically translated into our modeling language. Similar languages can also be designed for other types of contracts (e.g., social games). The proposed modeling language can be modified and extended to establish suitable mappings from new contract types to the classic mechanism model. With the new modeling language, the model extraction and property checking algorithms can be directly reused.

4 THE FAIRCON FRAMEWORK

In this section, we present the FairCon verification framework for smart contract fairness. Figure 5 shows the overall workflow of FairCon. The framework consists of three modules, namely, *model extraction*, *property checking*, and *fairness verification*.

The smart contract source code is first automatically instrumented according to user-provided annotations. At this stage, we consider a k-player bounded model, and the instrumented contract code contains a harness which orchestrates the interactions between the players and the target contract. The extraction of the mechanism model is powered by symbolic execution of the harness program, and an intermediate mechanism model is synthesized as a result.

In order to perform property checking, the intermediate mechanism model, along with the desired property, are encoded as an SMT formula, such that the formula is unsatisfiable if and only if the property holds with respect to the model. We use an SMT solver to check and may declare the property holds when the number of participants are bounded by k; otherwise, a counterexample is generated which disputes the property.

If we fail to find a counterexample in the bounded case, we may proceed to the fairness verification of the properties for unbounded number of participants. To do that, we modify the harness to account for an unlimited number of players, instrument it with program invariant as well as the desired properties as post-conditions, and rely on program verification tools to discharge the proof obligations. This either tells us that the property is successfully proved, or the validity of the property is still unknown, in which case we are only confident about the fairness for the bounded case.

4.1 Mechanism Model Extraction

To extract a mechanism model out of the smart contract source code, we first instrument the contract code with a harness program `MechanismHarness` shown in Fig. 6. The harness program orchestrates the interactions of k players with the target contract. This is achieved by declaring symbolic variables to represent the possible bid and valuation of each player, stored in the arrays "`BID`" (Line 3) and "`VALUE`" (Line 4), respectively. Then a for-loop (Lines 5–19) is used to simulate the actions performed by the k players. In smart contract, all players have to move sequentially since parallelization

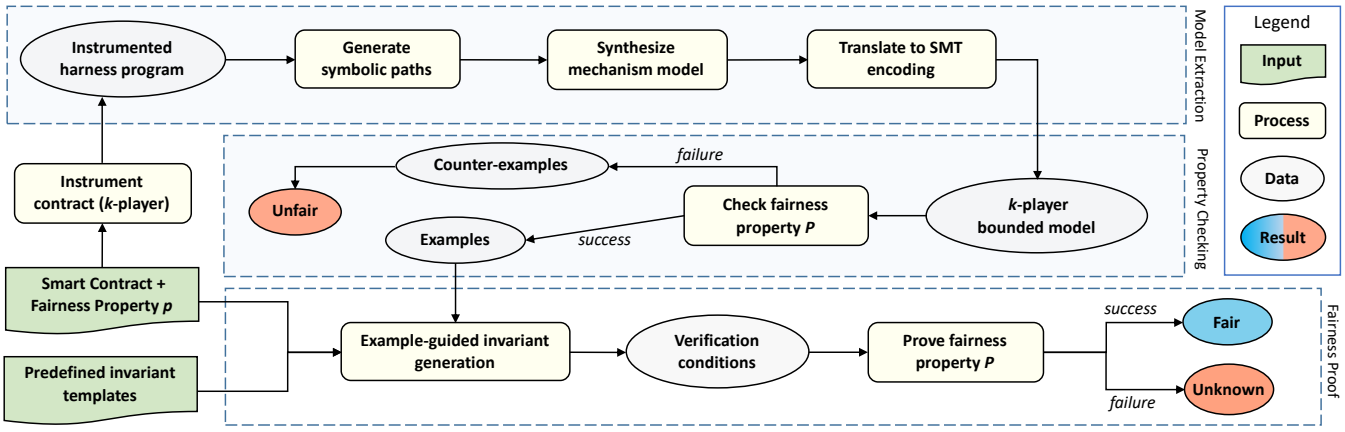

Figure 5: Workflow of the FairCon framework.

```
1  contract MechanismHarness {
2    // k-player bounded model
3    uint BID[k]; // symbolic values
4    uint VALUE[k]; // symbolic values
5    for (uint i=0; i<k; i++) {
6      // Example: msg.sender = i
7      require(@individual.id == toStr(i));
8      // Example: msg.value = BID[i]
9      require(@individual.bid == BID[i]);
10     // Example: bid() function inlined
11     @outcome;
12     // Example: ALLOCATE = highestBidder
13     ALLOCATE = @allocate;
14     // Example: PRICE = highestBid
15     PRICE = @price;
16     // Check loop invariant
17     // PRICE = max(BID[0..i]) ∧ ALLOCATE = arg max(BID[0..i])
18     assert( <invariant> );
19   }
20   // Check post condition
21   assert( <property> );
22 }
```

Figure 6: The harness program for mechanism model orchestration.

is not allowed. The ordering is not important, because the players are symmetric.

We rely on the annotations provided by users (e.g., Fig. 1) to construct the loop body, which triggers a move from one particular player. The variables controlling the player's identifier and bid value are assigned the corresponding symbolic values (Lines 7 and 9). In the case of Example 2.1, these variables are `msg.sender` and `msg.value`, respectively. Then the allocation function (e.g., `bid()` in Example 2.1) is inlined, and the resulting variables annotated by `@allocate` and `@price` are stored as symbolic expressions (Lines 13 and 15). There are also two placeholders at Lines 18 and 21, for assertions of loop invariant and post conditions, which will be described in Sect. 4.3.

We then run symbolic execution on the harness program to collect a set of feasible symbolic paths. Each symbolic path is represented in the form of "*Condition* ∧ *Effect*", where "*Condition*" and "*Effect*" are Boolean expressions in terms of the symbolic variables defined earlier (e.g., `BID[i]` and `VALUE[i]` in Fig. 6). Here, "*Condition*" represents the *path condition* which enables the execution of a particular program path; "*Effect*" represents the values of the resulting variables (e.g., `ALLOCATE` and `PRICE` in Fig. 6). We take all path conditions $Condition_j$, where the effect is successfully computed (i.e., not running into errors or reverts), and use the disjunction of them as the *assumption* of the model (i.e., "`assume` $\bigvee_j Condition_j$"). Similarly, we use the effects as the corresponding allocation and clear price functions. For example, in the mechanism model, we have "`allocate` $\bigvee_j(Condition_j \wedge Effect_j[ALLOCATE])$" and "`price` $\bigvee_j(Condition_j \wedge Effect_j[PRICE])$".

4.2 Bounded Property Checking

For property checking, given a mechanism model M and a property p, our goal is to construct a formula ϕ such that ϕ is unsatisfiable if and only if $M \models p$. With the semantic rules defined in Sect. 3.3, it is straightforward to obtain a formula encoding the allocation and clear price functions, i.e., $\varphi_M = d(\theta) \wedge t(\theta)$.

We illustrate the encoding of properties using the truthfulness as an example. Definition 3.2 states that a model M is truthful if and only if the truth-telling strategy performs no worse than any other strategies. Therefore, the high-level idea is to first encode the truthful and untruthful strategies separately for an arbitrary player, and then asserting that the utility of the player is higher when he/she acts untruthfully. The encoding of the truthfulness property p is shown as follows,

$$
\begin{aligned}
&\exists i.\forall j \cdot (i \neq j) \implies \\
&\quad (\varphi_M \wedge (bid_i = val_i) \wedge (bid_j = val_j)) && \text{(Truthful)} \\
&\wedge (\varphi_M[bid_i/bid_i', u_i/u_i'] \wedge (bid_i' \neq val_i)) && \text{(Untruthful)} \\
&\wedge (u_i < u_i'), && \text{(Utility)}
\end{aligned}
$$

where i is a generic player with utility u_i. The truthful scenario is when all players (including i) bid the same amount as their valuations, i.e., $bid_i = val_i$ and $bid_j = val_j$. The untruthful model is constructed by substituting the bid and utility variables of i with new copies bid_i' and u_i', and asserting $bid_i' \neq val_i$. Finally, we assert that $u_i < u_i'$. If p is satisfiable, we find a counterexample where an untruthful strategy performs better than the truthful strategy.

Otherwise, the truthful strategy is a dominant strategy for i. The encodings of other properties are similar.

4.3 Formal Proof for Unbounded Model

Consider the harness program in Fig. 6. The loop iterates k times to model k players joining in each iteration. We use induction to prove that the smart contract satisfies the fairness property for arbitrary number of players. Following the standard approach to proving program correctness, an invariant for the for-loop is required, i.e., `<invariant>` in Fig. 6. Normally, the loop invariant has to be derived manually. Fortunately, smart contracts are usually written in a more standard way than arbitrary programs, which makes it easier to generalize invariants for the same type of smart contracts, e.g., auctions considered in this work. A set of predefined invariant templates (according to specific types of contracts) have to be provided to the framework as inputs (Fig. 5). The followings are three common types of invariants required for auctions.

$$\texttt{ALLOCATE} = \arg\max(\texttt{BID}) \quad \text{(TopBidder)}$$

$$\texttt{PRICE} = \max(\texttt{BID}) \quad \text{(1st-Price)}$$

$$\texttt{PRICE} = \max(\texttt{BID} \setminus \{\texttt{BID}[\arg\max(\texttt{BID})]\}) \quad \text{(2nd-Price)}$$

The "TopBidder" invariant requires that the bidder with the highest bid becomes the winner. The "1st-Price" invariant requires that the highest bid is the clear price, while the "2nd-Price" invariant requires that the second highest bid is the clear price.

However, the invariant has to satisfy two conditions to conclude that the smart contract satisfies the fairness property. To elaborate on the conditions, we define the following notations. Let the harness program in Fig. 6 be abstracted as

$$\texttt{for(}\ \mathit{Cond}\ \texttt{)}\ \{\ S;\ \texttt{assert(Q)}\ \}\ \ \texttt{assert(P)},$$

where $Cond$ is the loop condition, Q and P are the `<invariant>` and `<property>`, respectively, and S represents the statements in the loop body before the assertion of the invariant. We also need the *strongest postcondition* [20] operator for discussion. The notation $sp(Pre, Stmt)$ represents the strongest postcondition after the program statement $Stmt$ is executed, provided the precondition Pre before the execution. For example, $sp(x = 2, \texttt{``x:=x+1''})$ would be $x = 3$.

We now formally define the validity for invariants. The invariant has to satisfy the following two conditions:

(1) the invariant is inductive, i.e., $sp(Cond \wedge Q, S) \implies Q$. Intuitively, it means that no matter how many iterations the loop performs, the invariant always holds.
(2) the invariant is strong enough to guarantee the fairness property, i.e., $Q \implies P$.

If the conditions are satisfied, we can conclude that the smart contract is fair for arbitrary number of players. The validity of the conditions can be checked by any program verification tools, and we use Dafny [38] in this work.

Notice that, in the search for valid invariant, we give up those violating the two validity conditions when analyzing mechanism models for bounded number of players (c.f. Sect. 4.1). That is, only those invariants that are valid for the bounded models are considered in proving for arbitrary number of players. More fairness properties may be proved when customized invariants are provided.

5 IMPLEMENTATION AND EVALUATION

We implement the proposed approach in our tool FairCon, which takes an annotated smart contract source code with the fairness property to be checked as input, extracts its mechanism model for a finite number of players (c.f. Sect. 4.1), and then automatically perform symbolic path analysis on the model (c.f. Sect. 4.2). If the fairness property is violated in one symbolic path, FairCon generates a counterexample related to that path. All the symbolic path analysis is achieved based on the Z3 SMT solver. If no counterexample is founded within a finite number of players, FairCon then tries to prove that the smart contract satisfies the fairness property based on induction with the set of predefined invariants. During the process, Dafny [38] is used to check the two validity conditions of invariants (c.f. Sect. 4.3) to establish the fairness proof. To explore the capability of our proposed approach in this paper, we evaluated FairCon to answer the research questions below.

- **RQ1**: How accurately does FairCon check fairness properties on smart contracts?
- **RQ2**: How efficient could FairCon be for mechanism model extraction and fairness property checking?
- **RQ3**: What are the common patterns for unfair smart contracts?

5.1 Experiment Setup

Many Ethereum smart contracts are token-based and derived from standard templates (e.g., there are 259,131 ERC-20 contracts on Etherscan), but such contracts have little fairness concerns. We collected 47,037 verified[2] smart contracts running on Ethereum from the Etherscan website, among which we found 129 contracts whose name or code contains keyword "auction". After code review for these smart contracts, we selected 20 typical auction contracts. The contracts that are not selected are either the presale contracts for auction or the auction contracts that end immediately after getting one bid, which are not within the scope of our fairness analysis in this paper. These selected contracts are actively in use, impacting many real users. For example, `CryptoRomeAuction`, `hotPotatoAuction`, and `Deed` are used to support popular DApps, and `Deed` has more than 1,469,061 transactions. In fact, the number of DApps on Ethereum is small (2.7K), compared with the total number of contract instances. Among the 20 selected auction contracts, we found that four auctions completely have the same structure. Finally, after removing duplicate or similar contracts, we selected 12 distinct auction contracts for our experiments. Apart from auction contracts, we also selected five voting smart contracts. So totally we have 17 public smart contracts (12 for auction and five for voting) for our experiments.

To find counterexamples, we set some configurations on mechanism models to be checked. For the auction mechanism model, there are three bidders, and the bid price and the valuation of bidders are arbitrary while allocation will be for one winner only. Similarly, for the voting mechanism model, we assume that five voters vote for two proposals as the basic configuration. Voter votes to any of proposals randomly, and his ballot could be reflected into the bid in our mechanism model. Voter has his own valuation for different proposals, and the winning proposal means the allocation. The

[2] A contract is labeled "verified" on `Etherscan` if its source code matches with the deployed version on Ethereum.

Table 2: Fairness checking on auction contracts.

Contracts	Properties				Time (seconds)	
	T	C	O	E	t_{model}	t_{check}
Auction1	✗	✗	✗	✗	7.96	0.11
Auction2	✗	✗	✗	✗	6.04	0.08
Auction3	✗	✗	✗	✗	2.34	0.08
AuctionItem	✗	✗	✓	✗	1.29	0.08
AuctionManager	✗	✗	✗	✗	1.61	0.10
AuctionMultipleGuaranteed	✗	✗	✗	✗	7.48	0.11
AuctionPotato	✗	✗	✗	✗	2.45	0.07
BetterAuction	✗	✗	✓	✗	1.58	0.08
CryptoRomeAuction	✗	✗	✗	✗	7.94	0.08
Deed	✓	✓	✗	✓	14.25	0.07
EtherAuction	✓	✓	✗	✗	8.43	0.08
hotPotatoAuction	✗	✗	✗	✗	5.48	0.09

Table 3: Fairness proving on fair auction contracts in Table 2.

Contracts	Allocation Inv.	Price Inv.	Proved Property
AuctionItem	TopBidder	1st-Price	O
Deed	TopBidder	2nd-Price	T, C, E
EtherAuction	N/A	N/A	–
BetterAuction	TopBidder	1st-Price	O

actual valuation of winning proposal or failing proposal is the sum of voters' valuation to that proposal. On Ethereum voting contracts are open to users, we assume voter cannot get any utility if the voter's supporting proposal is not the winning proposal. And the valuation of voter to proposal could be measured in two way. The first is that the valuation is mapped to real number. For instance, voter may prefer proposal A much more than any other voters prefer A. That is the situation where voters are heterogeneous. The second setting assigns 0 or 1 to valuation of voter to proposal. For instance, voter wants proposal A rather than proposal B. This simplified version could be applied to the situation where the voters are homogeneous. Under these settings, FairCon checks the four fairness properties at a given number of participants aiming to find counterexamples.

With the configurations for mechanism models, we spent 6 human hours to manually annotate mechanism components and instrument the harness in these smart contracts. Our experiments are conducted on Ubuntu 18.04.3 LTS desktop equipped with Intel Core i7 16-core and 32GB memory. We discuss the experiment results in the following subsections. The raw results and the replication package are available at: https://doi.org/10.21979/N9/0BEVRT.

5.2 Experiment Results

We now discuss the experiment findings in details.

Results for RQ1. To answer RQ1, we evaluated FairCon by the selected 17 smart contracts with the configurations mentioned in Sect. 5.1. Tables 2 and 4 show the results for fairness checking on auction and voting contracts, respectively. In Table 2, the first column shows the names of the contracts, and the middle four columns show the result for the four fairness properties: truthfulness (T), collusion-freeness (C), optimality (O), and efficiency (E),

Table 4: Fairness checking on voting contracts.

Contracts	Valuation: $\mathbb{R}$				Valuation: $\{0, 1\}$				t_{model}
	T	C	E	t_{check}	T	C	E	t_{check}	
Association	✗	✗	✗	0.35	✓	✓	✓	0.37	64.96
Ballot	✗	✗	✗	0.45	✓	✓	✓	0.81	69.73
Ballot-doc	✗	✗	✗	0.48	✓	✓	✓	0.56	126.14
HIDERA	✗	✗	✗	0.12	✓	✓	✓	0.15	52.23
SBIBank	✗	✗	✗	0.27	✓	✓	✓	0.69	56.59

respectively. The rightmost column indicates the time for mechanism model extraction (t_{model}) and for fairness property checking (t_{check}). Among the selected 12 contracts, four of them are found to be fair on at least one fairness property, while the remaining eight contracts are not fair for all the four fairness properties (with counterexamples generated). We had manually checked the generated counterexamples and confirmed that they are not false positives. Regarding the execution time, in our three-bidders experiments, model extraction time varied from 1.61 to 14.25 seconds because different contracts have different mechanism models to be extracted. Property checking is much faster than model extraction, which took around 0.1 seconds for each contract.

Table 3 shows the result of proving fairness properties for the 4 auction contracts that are fair on at least one fairness property, as shown in Table 2. The first column shows name of contract. The second and third columns show the invariant templates that are valid for proving fairness properties. The last column indicates which fairness property can be proved (T for truthfulness, C for collusion-freeness, O for optimality, and E for efficiency). The `AuctionItem` and `BetterAuction` contracts can be proved to satisfy the optimality property using the allocation invariant "TopBidder" together with the price invariant "1st-Price", which also confirms that they are first price auctions. The `Deed` contract satisfies the "TopBidder" and "2nd-Price" invariants, based on which, FairCon can prove three properties for `Deed`, namely, truthfulness, collusion-freeness, and efficiency. It also confirms that `Deed` is a second price auction.

The `EtherAuction` contract is shown in Fig. 7, which is an variant of second price auction for a designated bid price only. Line 13 requires a fixed new higher bid price to update the four variables `SecondHighestBid`, `SecondHighestBiddder`, `HighestBid`, and `HighestBiddder` in Lines 15–18, respectively. It turned out that none of our predefined invariants are valid to prove any fairness property. This is because `EtherAuction` adopts the strategy of fixed bid price for each round, which makes it similar to (but actually not) typical second price auctions.

Table 4 shows the result of property checking for the selected five voting smart contracts, each of which is for five voters and two proposals. The first column shows the names of contracts. The last column, t_{model}, shows the model extraction time (in seconds). The middle two large columns show the average property checking time (in seconds) for the three properties: truthfulness (T), collusion-freeness (C) and efficiency (E). We have two settings for the valuation component in our mechanism model. One is ranging over real numbers $\mathbb{R}$, while the other is ranging over $\{0, 1\}$. The reason of having two settings is that no contract was found fair regarding any property, as shows in the second large column of

```
contract EtherAuction {
  //Anyone can bid by calling this function and supplying the
  ↪ corresponding eth
  function bid() public payable {
    require(auctionStarted);
    require(now < auctionEndTime);
    require(msg.sender != auctioneer);
    // If sender is already the highest bidder, reject it.
    require(highestBidder != msg.sender);
    address _newBidder = msg.sender;
    uint previousBid = balances[_newBidder];
    uint _newBid = msg.value + previousBid;
    // Each bid has to be 0.05 eth higher
    if (_newBid  == highestBid + (5 * 10 ** 16)) return;
    // The highest bidder is now the second highest bidder
    secondHighestBid = highestBid;
    secondHighestBidder = highestBidder;
    highestBid = _newBid;
    highestBidder = _newBidder;
    latestBidTime = now;
    // Update the bidder's balance so they can later withdraw
    ↪ any pending balance
    balances[_newBidder] = _newBid;
  }
}
```

Figure 7: The EtherAuction Solidity source code.

Table 4, because the diverse $\mathbb{R}$ valuation of proposals brings the incentive for voters to lie and to conspire with others. In the $\{0, 1\}$ valuation setting, as shows in third large column of Table 4, all the five contracts are truthful, collusion-free, and efficient. This is because, if a voter lies, he/she gets at most zero worth utility, and thus has no incentive to lie. Based on Table 4, we can observe that fairness property may depend on the configuration of mechanism models. Different configurations may have different results on fairness property checking.

The checking time for the optimality property is not listed in Table 4 because smart contracts for voting do not have the component of transfer functions in our mechanism model so that optimality cannot be defined (c.f. Sect. 3.2). In addition, none of the predefined invariants are valid to prove that the five selected voting contracts are fair. We need to construct other valid invariants manually, which is one of our future works.

Answer to RQ1: Since there is a lack of ground truth, we manually investigated the cases and the results of FairCon were confirmed.

Results for RQ2. The time costs studied in RQ2 can be divided into two parts: model extraction and property checking. Overall, the model extraction takes much longer time and the property checking is efficient (taking less than one second for each case), as shown in Tables 2 and 4.

To explore the efficiency of FairCon further, we selected the `CryptoRomeAuction` contract to make performance experiments on FairCon. Figures 8 and 9 show the execution time for mechanism model extraction and fairness property checking when the number of bidders increases, respectively. In Fig. 8, the x-axis shows the number of bidders, while the y-axis shows the mechanism model extraction time in seconds. We can observe that model extraction time is nearly exponential to the number of bidders involved, which is reasonable because every participant is independent. When the number of bidders is under six, the model extraction time is less than 10 minutes, which is tolerable. Once the number of bidders goes beyond six, the time increases exponentially.

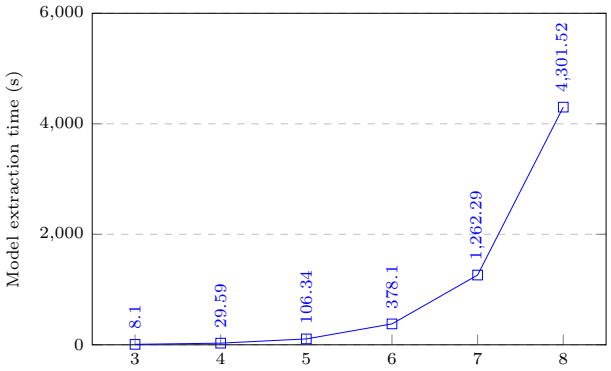

Figure 8: Model extraction time with increasing number of bidders.

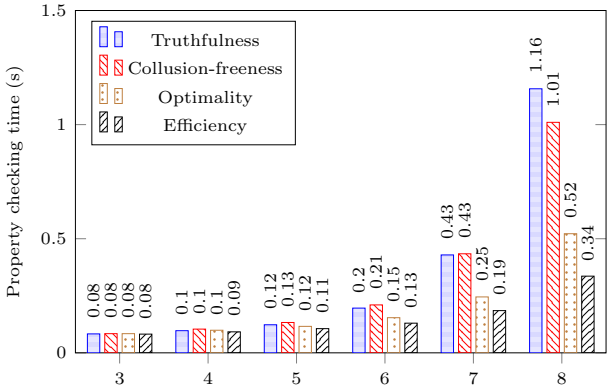

Figure 9: Property checking time with increasing number of bidders.

Figure 9 shows the property checking time, where the x-axis indicates the number of bidders, and the y-axis indicates the property checking time in seconds. We can observe the same trend as that in Fig. 8, i.e., the execution time is exponential to the number of bidders involved. However, property checking is much faster than model extraction since model extraction requires symbolic execution, which is heavy in computation. Mostly the checking time is less than one second. We can also observe that the checking time for truthfulness or collusion-freeness is at least doubled than that for optimality and efficiency. This is because truthfulness and collusion-freeness properties need to consider the strategy as well as the outcome spaces, while optimality and efficiency properties only have to consider the outcome space.

Running FairCon on a large number (i.e., k, which is theoretically unbounded) of players is impractical for symbolic execution. The results indicate that FairCon can finds counterexample(s) with small k (e.g., three to five), if the contract is indeed unfair (e.g., $k = 3$ in Table 2 and $k = 5$ in Table 4). Otherwise, as shown in Table 3, FairCon could use invariants observed during the small k runs in an attempt to prove that the fairness property holds for arbitrary values of k.

Answer to RQ2: Although model extraction is the bottleneck of FairCon, it is a one-time task and need not be performed for a large k. FairCon is efficient for fairness property checking.

Results for RQ3. Fairness issues in smart contracts are real. For example, `EtherAuction` (as shows in Fig. 7) was reported as a scam,[3] where bidders compete for one Ether by gradually increasing their bids. Our results confirmed its unfairness: although truthfulness and collusion-freeness holds for a bounded k, optimality and efficiency are violated. Based on our review of the subject contracts, we summarize some patterns below.

(1) Contracts implementing the first price auction and their variants do not satisfy the truthfulness property. For example, the aforementioned `BetterAuction` is implementing a typical open first price auction, where the top bidder has the incentive to lower his/her bid price but still remain the winner.
(2) Contracts implementing the first price auction and variants do not prevent against collusion. For example, `BetterAuction` does not satisfy the collusion-freeness property, since two bidders have the chance to lower the clear price and to be the winner, increasing their group utility.
(3) Contracts implementing the second price auction and their variants do not satisfy the optimality property. For example, `Deed` is one of the contracts implementing the second price auction. Since the clear price is the second highest bid, the contract may not be optimal with a potential decrease in total revenue.
(4) Contracts implementing the first price auction and their variants are not efficient. This is because first price auctions are untruthful, and the winner may not be the one who has the highest valuation of the item.

We envision that fairness checking be included as a part of the common-practice validation process to improve users' confidence towards DApps powered by smart contracts. We hope the fairness issues in smart contracts can be mitigated by alerting developers and users these common patterns.

5.3 Threats to Validity

Our evaluation results are subject to common threats to validity.

Lack of ground truth. It lacks ground truth for the contracts and properties we studied. Two of the authors manually inspected the subjects and our results independently, which took around half an hour for each contract. We confirmed that the counterexamples provided by our tool are valid.

External validity. The types of contracts and properties considered in this work are limited. Our findings may not be generalized to other cases. The DApps implemented with smart contracts usually follow typical patterns, mainly due to the limitations on language syntax and considerations on gas consumption. We believe that other types of game-like contracts would behave similarly.

[3] https://hackernoon.com/take-your-chances-at-the-ether-auction-game-30f9df1ec80b

6 RELATED WORK

Our work is closely related to the following research areas: (1) the functional correctness and security analysis of smart contracts, (2) the verification of fairness properties in traditional software systems, (3) and mechanism design as well as game theory.

6.1 Smart Contract Analysis and Verification

Since smart contract applications are often used to manage a large sum of funds, detection of security flaws in smart contracts received a lot of attention. The violation of important security properties leads to many well-known smart contract vulnerabilities [54]. For example, a smart contract which fails to check the return value of a (possibly failed) external call operation, has the *unchecked call* vulnerability [25, 48]. The execution logic of a smart contract that is not independent of environmental variables, e.g., the block timestamp, is prone to *dependence manipulation* [59], including the *timestamp dependency* vulnerability [39]. If the business logic of a smart contract depends on its mutable state parameters, such as balance and storage, then it has the *transaction-ordering dependence* problem. A smart contract is *reentrant*, if provided with enough gas, an external callee can repeatedly call back into it within a single transaction. Missing permission checks for the execution of a `transfer` or a `selfdestruct` operation make a smart contract *prodigal* and *suicidal*, respectively [44]. Absence of proper checks for arithmetic correctness make Ethereum contracts prone to *integer and batch overflow/underflow* [22, 51]. Furthermore, the progress of a smart contract can be compromised by gas-exhaustive code patterns [16, 24].

To address these security issues, Ellul et al. developed a runtime verification technique, ContractLarva [21], to rule out certain unsafe behaviors during the execution of smart contracts. Oyente [2, 39] is one of the first to detect smart contract vulnerabilities using symbolic execution. ContractFuzzer [31] is among the early fuzz testing tools and Mythril [1] is a well-known security analysis tool which combines symbolic execution and taint analysis to detect nearly 30 classes of vulnerabilities. There are many other tools [14, 34, 50, 52, 55, 57, 58] designed for the similar purpose.

Another popular direction is using formal techniques to ensure the functional correctness of smart contracts. Bhargavan et al. devised a functional programming language, named F^* [12], to facilitate the formal verification of Ethereum smart contracts. Based on F^*, Grishchenko et al. presented the first complete small-step semantics of EVM bytecode [25]. Hildenbrandt et al. presented an executable formal semantics for the Ethereum platform, named KEVM [27], based on which, Park et al. [46] presented a deductive verification tool, capable of verifying various high-profile and safety-critical contracts. Jiao et al. developed the operational formal semantics for the Solidity programming language, named K-Solidity [32, 33]. Abdellatif et al. [7] formalized blockchain and users' behaviors to verify properties about their interactions using statistical model checking. Nehai et al. [43] applied model checking to verify smart contracts from the energy market field.

Most of the smart contract analyses mentioned above focus on finding bugs or security vulnerabilities, which highlight mismatches between contract developers' expectations and how the contract code work; whereas the fairness issues considered in our paper

highlight mismatches between the contract users' expectations and the actual implementation of the game rules.

6.2 Fairness Checking in Software Systems

We believe that fairness should be considered a software quality attribute—among functional correctness, security, privacy, etc.—one needs to consider throughout the software development process. Smart contract is an emerging type of software application with often multiple interacting participants, where fairness becomes a lot more relevant.

The problem of *algorithmic fairness* is considered in many modern decision-making programs [8, 19, 62], either learned from data or created by experts. The term "fairness" can be subjective depending on the actual contexts. Verma and Rubin [56] collected definitions of fairness from different software domains and explained the rationales behind these definitions. From the software specification and verification perspective, Albarghouthi et al. [8] treated decision-making algorithms as probabilistic programs and proposed to verify formally defined fairness properties on a wide class of programs. D'Antoni et al. [9] introduced the concept of *fairness-aware programming* and presented a specification language and runtime monitoring technique which allow programmers to specify fairness properties in their code and enforce the properties during executions. In general, the fairness definition in such decision-making programs is that the program shows no bias towards certain groups of users. There is little consideration in terms of the interactions, interests, and conflicts between users and programs, or between users and users.

With regard to smart contracts, Bartoletti et al. [10] found through a survey that nearly 0.05% of the transactions on Ethereum could be owing to Ponzi schemes. Chen et al. [17] identified patterns in contract applications implementing Ponzi schemes, and built a classifier to detect suspicious schemes using data mining and machine learning. Such contracts can be considered violating fairness properties, in the sense that not all participants have the same chance of gaining profits. In this work, we expand the notion of fairness in smart contracts to include any properties expressible in mechanism models.

6.3 Mechanism Design and Game Theory

Mechanism design has been well studied in the economic domain [29, 36, 37, 40]. These works offer the theoretical foundation for our model extraction and fairness verification. Many fairness properties we used in this paper are also inspired by them. Maskin [40] articulated some important concepts, such as *outcomes* and *social goals*, in implementation theory, which is a part of mechanism design. He offered a well-defined example to show how to achieve social goals. Jackson [29] presented mechanism theory in a full view and provided formal definitions to many concepts belonging to this domain, while Klemperer [36] introduced the most fundamental concepts for auction and carried out a thorough analysis of optimal auctions, the equivalence theorem, and marginal revenues. Lehmann [37] revealed how to exploit truth revelation in realizing approximately efficient combination auction which emphasized the co-exist problem of *optimal* auction and *efficient* auction.

Mechanism design and game theory were also applied on the smart contract design. Hahn et al. [26] implemented a Vickrey second price auction on a smart contract to setup and operate a market of energy exchanges. Similarly, Chen et al. [18] provided an e-auction mechanism based on blockchain to ensure confidentiality, non-repudiation, and unchangeability of the electronic sealed bid. CReams [61] implemented a collusion-resistant k-Vickery auction. Galal and Youssef [23] presented a smart contract protocol for a succinctly verifiable sealed-bid auction on the Ethereum blockchain to protect bidders' privacy. Mccorry et al. [41] proposed the first implementation of a decentralized and self-tallying internet voting protocol using smart contract to guarantee secure e-voting. Bigi et al. [13] combined game theory and formal models to analyze and validate a decentralized smart contract protocol, named *DSCP*, and used game theory to analyze users' behavior. Chatterjee et al. [15] studied two-player zero-sums games and performed a quantitative analysis of players' worst case utilities. These works employ certain levels of fairness analyses, mostly manual, on some one-off applications. In contrast, we provide a more general framework with maximal automation support.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed an approach to analyze fairness properties of smart contracts. We implemented FairCon to automatically extract mechanism models from smart contracts with user-provided annotations, and experimentally evaluated it on 17 real-world auction and voting contracts. The experiment results indicate that FairCon is effective in detecting property violations and able to prove fairness for common types of contracts.

In the future, we would like to apply FairCon to other types of smart contracts beyond auction and voting. It can also be extended to check for other types of fairness properties that are critical in maintaining the integrity of blockchain applications.

ACKNOWLEDGMENTS

This research is partly supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (MOE2018-T2-1-068) and the National Research Foundation, Prime Ministers Office Singapore under its National Cybersecurity R&D Program (NRF2018NCR-NCR005-0001).

REFERENCES

[1] 2019. Mythril. https://github.com/ConsenSys/mythril. A Security Analysis Tool for EVM Bytecode.
[2] 2019. Oyente. https://github.com/melonproject/oyente. An Analysis Tool for Smart Contracts.
[3] 2020. DApp Statistics. https://www.stateofthedapps.com/stats.
[4] 2020. EIP-1202: Voting Standard. https://eips.ethereum.org/EIPS/eip-1202.
[5] 2020. Etherscan. https://etherscan.io.
[6] 2020. Interface for Blind Auctions (Draft). https://github.com/ethereum/EIPs/pull/1815.
[7] Tesnim Abdellatif and Kei-Léo Brousmiche. 2018. 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)*. IEEE, 1–5.
[8] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proceedings of the ACM on Programming Languages* 1 (2017), 80:1–80:30.
[9] Aws Albarghouthi and Samuel Vinitsky. 2019. Fairness-Aware Programming. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*. 211–219.

[10] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2019. Dissecting Ponzi Schemes on Ethereum: Identification, Analysis, and Impact. *Future Generation Computer Systems* (2019).

[11] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2020. Dissecting Ponzi Schemes on Ethereum: Identification, Analysis, and Impact. *Future Generation Computer Systems* 102 (2020), 259 – 277. https://doi.org/10.1016/j.future.2019.08.014

[12] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 91–96.

[13] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. 2015. Validation of Decentralised Smart Contracts through Game Theory and Formal Methods. In *Programming Languages with Applications to Biology and Security*. Springer, 142–161.

[14] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. 2018. sCompile: Critical Path Identification and Analysis for Smart Contracts. *arXiv preprint arXiv:1808.00624* (2018).

[15] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *European Symposium on Programming*. Springer, Cham, 739–767.

[16] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-Optimized Smart Contracts Devour Your Money. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*. 442–446. https://doi.org/10.1109/SANER.2017.7884650

[17] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. 2018. Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology. In *Proceedings of the 2018 World Wide Web Conference*. International World Wide Web Conferences Steering Committee, 1409–1418.

[18] Yi-Hui Chen, Shih-Hsin Chen, and Iuon-Chang Lin. 2018. Blockchain Based Smart Contract for Bidding System. In *2018 IEEE International Conference on Applied System Invention (ICASI)*. IEEE, 208–211.

[19] Anupam Datta, Shayak Sen, and Yair Zick. 2016. Algorithmic Transparency via Quantitative Input Influence: Theory and Experiments with Learning Systems. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 598–617.

[20] Edsger W. Dijkstra and Carel S. Scholten. 1990. Predicate Calculus and Program Semantics. *Texts and Monographs in Computer Science* (1990).

[21] Joshua Ellul and Gordon J Pace. 2018. Runtime Verification of Ethereum Smart Contracts. In *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 158–163.

[22] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise Attack Synthesis for Smart Contracts. *arXiv preprint arXiv:1902.06067* (2019).

[23] Hisham S Galal and Amr M Youssef. 2018. Succinctly Verifiable Sealed-Bid Auction Smart Contract. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 3–19.

[24] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* (2018), 116.

[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[26] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. 2017. Smart Contract-Based Campus Demonstration of Decentralized Transactive Energy Auctions. In *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*. IEEE, 1–5.

[27] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. 2017. *KEVM: A Complete Semantics of the Ethereum Virtual Machine*. Technical Report.

[28] EOS IO. 2017. EOS. IO Technical White Paper. *EOS. IO (accessed 18 December 2017) https://github. com/EOSIO/Documentation* (2017).

[29] Matthew O Jackson. 2014. Mechanism Theory. *Available at SSRN 2542983* (2014).

[30] Mamata Jenamani, Yuhui Zhong, and Bharat Bhargava. 2007. Cheating in Online Auction–Towards Explaining the Popularity of English Auction. *Electronic Commerce Research and Applications* 6, 1 (2007), 53–62.

[31] Bo Jiang, Ye Liu, and WK Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 259–269.

[32] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1695–1712.

[33] Jiao Jiao, Shang-Wei Lin, and Jun Sun. 2020. A Generalized Formal Semantic Framework for Smart Contracts. In *2020 International Conference on Fundamental Approaches to Software Engineering (FASE)*. 75–96.

[34] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *The Network and Distributed System Security Symposium*.

[35] James C King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[36] Paul Klemperer. 1999. Auction Theory: A Guide to the Literature. *Journal of Economic Surveys* 13, 3 (1999), 227–286.

[37] Daniel Lehmann, Liadan Ita Oćallaghan, and Yoav Shoham. 2002. Truth Revelation in Approximately Efficient Combinatorial Auctions. *Journal of the ACM (JACM)* 49, 5 (2002), 577–602.

[38] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

[39] 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*. ACM, 254–269.

[40] Eric S Maskin. 2008. Mechanism Design: How to Implement Social Goals. *American Economic Review* 98, 3 (2008), 567–76.

[41] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. 2017. A Smart Contract for Boardroom Voting with Maximum Voter Privacy. In *International Conference on Financial Cryptography and Data Security*. Springer, 357–375.

[42] Satoshi Nakamoto et al. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008).

[43] Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. 2018. Model-Checking of Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 980–987.

[44] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 653–663.

[45] Noam Nisan and Amir Ronen. 2001. Algorithmic Mechanism Design. *Games and Economic behavior* 35, 1-2 (2001), 166–196.

[46] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. 2018. 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*. ACM, 912–915.

[47] PeckShield. 2019. *EOS: Transaction Congestion Attack*. https://medium.com/@peckshield/eos-transaction-congestion-attack-attackers-could-paralyze-eos-network-with-minimal-cost-9adfb4d16c82

[48] Daniel Perez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? (feb 2019). arXiv:1902.06710 http://arxiv.org/abs/1902.06710

[49] David Siegel. 2016. *Understanding the DAO Attack*. https://www.coindesk.com/understanding-dao-hack-journalists

[50] SmartDec 2019. *SmartCheck*. SmartDec. https://tool.smartdec.net

[51] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 825–841.

[52] Sofware Reliability Lab 2019. *Securify*. Sofware Reliability Lab. https://securify.ch/

[53] Solidity 2018. Solidity. https://solidity.readthedocs.io/en/v0.5.1/.

[54] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2020. A Survey of Smart Contract Formal Specification and Verification. arXiv:2008.02712 [cs.SE]

[55] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 67–82.

[56] Sahil Verma and Julia Rubin. 2018. Fairness Definitions Explained. In *IEEE/ACM International Workshop on Software Fairness (FairWare)*. 1–7. https://doi.org/10.1145/3194770.3194776

[57] Haijun Wang, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2019. Oracle-Supported Dynamic Exploit Generation for Smart Contracts. arXiv:1909.06605 [cs.CR]

[58] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. Vultron: Catching Vulnerable Smart Contracts Once and for All. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*. IEEE Press, 1–4.

[59] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 189 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360615

[60] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

[61] Shuangke Wu, Yanjiao Chen, Qian Wang, Minghui Li, Cong Wang, and Xiangyang Luo. 2018. CReam: A Smart Contract Enabled Collusion-Resistant E-auction. *IEEE Transactions on Information Forensics and Security* 14, 7 (2018), 1687–1701.

[62] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. 2013. Learning Fair Representations. In *International Conference on Machine Learning*. 325–333.