Towards Smart Contract Fairness Verification, Access Control and Model-Based Testing

Liu Ye

School of Computer Science & Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

2021

List of Author's Publications[1]

Journal Articles

- Wang, Haijun and **Liu, Ye** and Li, Yi and Lin, Shang-Wei and Artho, Cyrille and Ma, Lei and Liu, Yang, "Oracle-supported dynamic exploit generation for smart contracts" *IEEE Transactions on Dependable and Secure Computing*, 2020.

Conference Proceedings

- **Liu, Ye** and Li, Yi and Lin, Shang-Wei and Zhao, Rong, "Towards automated verification of smart contract fairness" in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.* Accepted.

- **Liu, Ye** and Li, Yi and Lin, Shang-Wei and Yan, Qiang, 'ModCon: a model-based testing platform for smart contracts" in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.* Accepted.

[1]The superscript * indicates joint first authors

Chapter 1

Introduction

1.1 Motivation and Challenge

The blockchain technology has been booming in recent years, since the introduction of Bitcoin [Nakamoto et al.(2008)] by Nakamoto in 2008. The blockchain networks can be broadly categorized into the permissionless and permissioned blockchains, where the former is open to the public (e.g., Bitcoin [Nakamoto et al.(2008)] and Ethereum [Wood(2014)]) and the latter is only accessible to trusted groups or private institutions (e.g., Hyperledger Fabric [hyp(2020)] and FISCO BCOS [fis(2020)]). The distributed and tamper-resistant nature of blockchain has made it the perfect platform for hosting smart contracts. Smart contracts are stateful computer programs running on blockchain platforms to manage large sums of cryptocurrency, govern and carry out transactions of assets between multiple parties. Ethereum [Wood(2014)] and EOS [IO(2017)] are the most popular permissionless blockchain platforms which support smart contracts applicable in many areas. As of Nov 2020, there are over 1.5 million smart contracts deployed on Ethereum, which is a 100 fold increase since just two years ago. These smart contracts have enabled about 3.1K decentralized applications (DApps) [dap(2020)] serving around 80K daily users on finance, health, governance, gambling, games, etc.

The security of smart contracts has attracted great attention, ever since their adoption in the management of massive cryptocurrency transactions. The most notorious attack is that malicious attackers exploited the *reentrancy vulnerability* in the DAO contract on Ethereum [Siegel(2016)], which resulted in a loss of $60

million worth in 2016. EOS gambling games were intensively hacked in 2019 using a technique called the *transaction congestion attack* [PeckShield(2019)] and led to significant asset loss of smart contract applications such as `EOS.WIN` and `EOSPlay`. The root cause of these attacks is that certain security vulnerabilities introduced during contract development are exploited by malicious parties, causing a loss for the contract owners (and possibly other honest participants). Most vulnerabilities subject to programming errors, indicating a mismatch between the contract developers' expectations and how the contract code actually works. Therefore, much research has been dedicated to preventing attacks, discovering and mitigating such vulnerabilities .

However, little work has been done on more enterprise smart contract applications for its great complexity. The contract execution on permissionless chains is bounded by limited resource such that smart contracts are often kept simple. For example, on Ethereum, user has to pay miners (who in charge of executing transactions) a certain amount of "gas" (cryptocurrency on Ethereum) as the transaction fee to deploy or interact with contract, which is mainly decided by the complexity of the contract (e.g., up to $15 in fees [gas(2020)]). Therefore, most permissionless blockchains are mainly used for crytocurrency exchange (e.g., ERC Token and DeFi applications). In contrast, the permissioned blockchains aim to create real value. For instance, FISCO BCOS has been successfully adopted in areas such as government and judicial services, supply chain, finance, health care, copyright management, education, transportation, and agriculture [fis(2020)]. The smart contracts powering these enterprise applications are more sophisticated and often demonstrate strong stateful behaviors. However, existing testing and analysis tools [Jiang et al.(2018)Jiang, Liu, and Chan, oye(2019), Sof(2019), Sma(2019), Wang et al.(2019)Wang, Li, Lin, Ma, and Liu] target only Ethereum smart contracts and mainly focus on their security issues, which will not work well in these stateful enterprise smart contracts due to lacking understanding of system behaviours, absence of test oracles and missing measurement of test adequacy. The aforementioned enterprise smart contracts on permissioned blockchains are usually specified suitable for model based testing (MBT). However, for most smart contracts on permissionless blockchains, system behaviours and requirements are not specified thus making existing MBT tools impractical for finding logic bugs. Meanwhile, popular Ethereum DApps usually involve many users in thousands or even millions of transactions [Eth(2020)] that occurred in the past, from which system behaviours

(e.g., user behavior) and implementation requirements (e.g., program invariants) as well as system access control models can be learned to some extent.

Compared to the security, the *fairness* issues of smart contracts have not yet attracted much attention. Smart contracta are unfair to certain participants if there is a mismatch between the participants' expectations and the actual implementation of the game rules. Although 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, we would like to focus more on the fairness issues introduced by the poor design of the contracts instead, which are orthogonal to the security issues. For example, smart contracts could be "Ponzi schemes" [Bartoletti et al.(2020)Bartoletti, Carta, Cimoli, and Saia] even claiming to be "social games" with a promised 20% return for any investment. 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 [Wu et al.(2018)Wu, Chen, Wang, Li, Wang, and Luo]. The fairness issues mostly reside in contract logic: some are design defects, while the rest are careless mistakes. This makes the detection of such issues particularly challenging, since there is no hope in identifying general predefined patterns for every different cases. Meanwhile, 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 spare efforts 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.

To conclude, in this paper, our research aims are twofold: Our research aims are twofold: (1) Ensure smart contract security by performing model based testing and reverse engineering access control models for stateful enterprise smart contracts. (2) Verify smart contract fairness on mechanism models extracted from smart contract that reflect the interactions between users and contract.

1.2 Main Work and Contribution

Main Work and Contributions. Our main works and 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. 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 designed ModCon allowing users to provide a test model for smart contracts. The model is used to specify the state definitions, expected transition relations, pre/post conditions for each transition, invariants, and the mapping from the model to the contract code. Given the test model, the testing process can also customized by choosing from different coverage strategies and test prioritization options. ModCon then generates tests with the goal that exercise as many system behaviors as possible while prioritizing on cases of particular interests.
- We proposed a novel approach to apply history transactions for reverse engineering access control models of smart contract. We extracted role-based access control models from history transactions and studied and verified two types of access constraints: (1) separation of duty constraint, and (2) cardinality constraint.

Chapter 2

Toward Automated Verification of Smart Contract Fairness

2.1 Introduction

The blockchain technology has been developed rapidly in recent years, since the introduction of Bitcoin [Nakamoto et al.(2008)] 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 [Wood(2014)] and EOS [IO(2017)] 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) [dap(2020)] 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 [Siegel(2016)] 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 [PeckShield(2019)] 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" [Bartoletti et al.(2020)Bartoletti, Carta, Cimoli, and Saia]. 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 [Wu et al.(2018)Wu, Chen, Wang, Li, Wang, and Luo]. 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 [Jackson(2014),

Nisan and Ronen(2001)]), 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* [King(1976)] 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` [Eth(2020)]. 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 2.2 illustrates the workflow of FairCon with an example. Section 2.3 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. 2.4. **??** gives details on the implementation and presents the evaluation results. **????** compare FairCon with the related work and conclude the paper, respectively.

2.2 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 2.1 shows a simplified Ethereum smart contract, named `CryptoRomeAuction`, written in Solidity [Solidity()], taken from `Etherscan`.[1] The contract implements a variant of open English auction for a blockchain-based

[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 2.1: The CryptoRomeAuction Solidity source code.

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.

Threats to Contract Fairness. One way that `CryptoRomeAuction` can become unfair to the participants is through the so called *shill bidding* [Jenamani et al.(2007)Jenamani, Zho a shill tries to escalate the price without any intention of buying the item. This can

Table 2.1: 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

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 2.1 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 2.1 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

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

FIGURE 2.2: The mechanism model of CryptoRomeAuction with three bidders.

bids four, and they divide the profit equally among themselves. Each of them gains one unit of utility as a result.

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. 2.1. 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 [King(1976)] 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. 2.2. 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. 2.3. 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. 2.2 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. 2.4.2. 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 [Leino(2010)], to perform the automated verification.

2.3 The Analysis Framework for Smart Contract Fairness

Provide a general introduction and overview of the materials/methods and supply essential background information

In this section, we first provide necessary background and definitions on mechanism models and fairness properties well studied in the mechanism design literature [Jackson(2014), Nisan and Ronen(2001)]. Then we give the abstract syntax and semantics of our mechanism modeling language to support automated model construction and property checking.

2.3.1 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: $u_i(\hat{\theta}) = v_i(\hat{\theta}, \theta_i) - t_i(\hat{\theta})$.

2.3.2 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. 2.3.1 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 2.1 (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 2.2 (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 2.3 (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 2.4 (Optimality)**.** We say a mechanism is optimal if and only if its transfer function achieves maximum total net profit, i.e., $\forall \hat{\theta} \in \Theta \forall t' \cdot \sum_i t_i(\hat{\theta}) \geq \sum_i t'_i(\hat{\theta})$.

```
<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 2.3: Syntax of the auction/voting mechanism model.

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 2.5 (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.

2.3.3 Mechanism Modeling Language

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

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

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

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

FIGURE 2.4: Semantic rules of the auction/voting model.

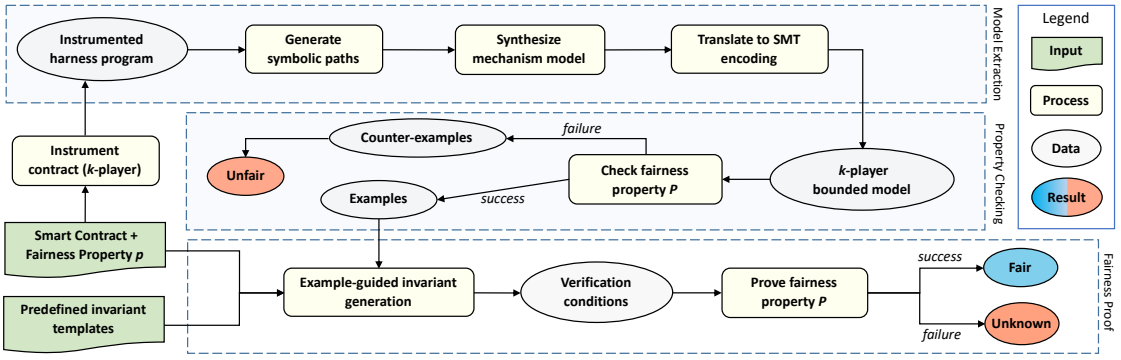

FIGURE 2.5: Workflow of the FAIRCON framework.

We define an abstract syntax of the mechanism modeling language, which is applicable to both auction and voting. Figure 2.3 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. 2.4. 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) [erc(2020a)] and ERC-1815 (blind auctions, under review) [erc(2020b)] 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.

2.4 The FairCon Framework

Restate the purpose of the work and provide specific and precise details about source of materials and methods

In this section, we present the FairCon verification framework for smart contract fairness. Figure 2.5 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

```
contract MechanismHarness {
  // k-player bounded model
  uint BID[k]; // symbolic values
  uint VALUE[k]; // symbolic values
  for (uint i=0; i<k; i++) {
    // Example: msg.sender = i
    require(@individual.id == toStr(i));
    // Example: msg.value = BID[i]
    require(@individual.bid == BID[i]);
    // Example: bid() function inlined
    @outcome;
    // Example: ALLOCATE = highestBidder
    ALLOCATE = @allocate;
    // Example: PRICE = highestBid
    PRICE = @price;
    // Check loop invariant
    // PRICE = max(BID[0..i]) ∧ ALLOCATE = arg max(BID[0..i])
    assert( <invariant> );
  }
  // Check post condition
  assert( <property> );
}
```

FIGURE 2.6: The harness program for mechanism model orchestration.

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.

2.4.1 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. 2.6. 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 is not allowed. The ordering is not important, because the players are symmetric.

We rely on the annotations provided by users (e.g., Fig. 2.1) 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. 2.4.3.

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* $\wedge$ *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. 2.6). 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. 2.6). 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` $\bigvee_j(Condition_j \wedge Effect_j[PRICE])$".

2.4.2 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. 2.3.3, 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 2.2 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.

2.4.3 Formal Proof for Unbounded Model

Consider the harness program in Fig. 2.6. 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. 2.6. 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. 2.5). 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. 2.6 be abstracted as

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

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* [Dijkstra and Scholten(1990)] operator for discussion. The notation $sp(Pre, Stmt)$ represents the strongest post-condition 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., $sp(\mathit{Cond} \wedge Q, S) \implies Q$. 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 [Leino(2010)] in this work.

Indicate the appropriate care was taken 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. 2.4.1). 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.

Bibliography

[Nakamoto et al.(2008)] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008. 1, 5

[Wood(2014)] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014. 1, 5

[hyp(2020)] Open, proven, enterprise-grade dlt. `https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf`, 2020. 1

[fis(2020)] FISCO BCOS. `https://fisco-bcos.org/`, 2020. 1, 2

[IO(2017)] EOS IO. Eos. io technical white paper. *EOS. IO (accessed 18 December 2017) https://github. com/EOSIO/Documentation*, 2017. 1, 5

[dap(2020)] Dapp statistics. `https://www.stateofthedapps.com/stats`, 2020. 1, 5

[Siegel(2016)] David Siegel. *Understanding the DAO Attack*, 2016. URL `https://www.coindesk.com/understanding-dao-hack-journalists`. 1, 5

[PeckShield(2019)] PeckShield. *EOS: Transaction Congestion Attack*, 2019. URL `https://medium.com/@peckshield/eos-transaction-congestion-attack-attackers-could-paralyze-eos-network-with-`
2, 6

[gas(2020)] Ethereum transaction fees fall by 75% as congestion eases. `https://cointelegraph.com/news/ethereum-transaction-fees-fall-by-75-as-congestion-eases`, 2020. 2

[Jiang et al.(2018)Jiang, Liu, and Chan] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018. 2

[oye(2019)] Oyente. `https://github.com/melonproject/oyente`, 2019. An Analysis Tool for Smart Contracts. 2

[Sof(2019)] *Securify*. Sofware Reliability Lab, 2019. URL `https://securify.ch/`. 2

[Sma(2019)] *SmartCheck*. SmartDec, 2019. URL `https://tool.smartdec.net`. 2

[Wang et al.(2019)Wang, Li, Lin, Ma, and Liu] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. Vultron: catching vulnerable smart contracts once and for all. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, pages 1–4. IEEE Press, 2019. 2

[Eth(2020)] Etherscan. `https://etherscan.io`, 2020. 2, 7

[Bartoletti et al.(2020)Bartoletti, Carta, Cimoli, and Saia] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: Identification, analysis, and impact. *Future Generation Computer Systems*, 102:259 – 277, 2020. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2019.08.014. URL `http://www.sciencedirect.com/science/article/pii/S0167739X18301407`. 3, 6

[Wu et al.(2018)Wu, Chen, Wang, Li, Wang, and Luo] Shuangke Wu, Yanjiao Chen, Qian Wang, Minghui Li, Cong Wang, and Xiangyang Luo. Cream: a smart contract enabled collusion-resistant e-auction. *IEEE Transactions on Information Forensics and Security*, 14(7):1687–1701, 2018. 3, 6

[Jackson(2014)] Matthew O Jackson. Mechanism theory. *Available at SSRN 2542983*, 2014. 7, 12

[Nisan and Ronen(2001)] Noam Nisan and Amir Ronen. Algorithmic mechanism design. *Games and Economic behavior*, 35(1-2):166–196, 2001. 7, 12

[King(1976)] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. 7, 11

[Solidity()] Solidity. Solidity. `https://solidity.readthedocs.io/en/v0.5.1/`, 2018. 8

[Jenamani et al.(2007)Jenamani, Zhong, and Bhargava] Mamata Jenamani, Yuhui Zhong, and Bharat Bhargava. Cheating in online auction–towards explaining the popularity of english auction. *Electronic Commerce Research and Applications*, 6(1):53–62, 2007. 9

[Leino(2010)] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3642175104. 12, 22

[erc(2020a)] Eip-1202: Voting standard. `https://eips.ethereum.org/EIPS/eip-1202`, 2020a. 17

[erc(2020b)] Interface for blind auctions (draft). `https://github.com/ethereum/EIPs/pull/1815`, 2020b. 17

[Dijkstra and Scholten(1990)] Edsger W. Dijkstra and Carel S. Scholten. Predicate calculus and program semantics. *Texts and Monographs in Computer Science*, 1990. 21