NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

On Security and Reliability of Smart Contracts

Liu Ye

School of Computer Science & Engineering

Qualifying Examination Report Submitted to the Nanyang Technological University
for the Confirmation for Admission to the Degree of Doctor of Philosophy

2021

Abstract

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. The security of smart contracts has attracted great attention, ever since their adoption in the management of massive cryptocurrency transactions. However, little work has been done on stateful smart contract applications for its great complexity. The main challenge lies that most smart contracts are not well specified. Compared to the security, the fairness issues of smart contracts have not yet attracted much attention. However, there is a likely mismatch between the participants' expectations and the actual implementation of contracts.

In this progress report, several solutions are proposed to address these challenges. An automated verification framework of smart contract fairness is used to help find the mismatch between the user expectations and the contract implementation where contracts are abstracted as games, i.e., mechanism models and the fairness properties originate from game theory as well. For enterprise smart contracts, they are well specified. A model-based testing framework is provided to facilitate the testing of these smart contracts. To overcome the missing of contract specification, our future work aims to learn a likely specification of smart contracts and find user permission bugs via time travel on transaction history. A big picture of the framework to enforce both security and fairness of smart contracts also remains to do in the future.

Contents

Abstract **iii**

List of Figures **vii**

List of Tables **ix**

List of Author's Publications **xi**

1 Introduction **1**
- 1.1 Smart Contract . . . 1
- 1.2 Motivation and Challenge . . . 2
- 1.3 Research Objective . . . 4
- 1.4 Main Work and Contribution . . . 5

2 Fairness Verification **7**
- 2.1 Introduction . . . 7
- 2.2 FairCon by Example . . . 8
- 2.3 The Analysis Framework for Smart Contract Fairness . . . 12
  - 2.3.1 Smart Contracts as Mechanism Models . . . 12
  - 2.3.2 Fairness Properties . . . 13
  - 2.3.3 Mechanism Modeling Language . . . 15
- 2.4 The FairCon Framework . . . 17
  - 2.4.1 Mechanism Model Extraction . . . 18
  - 2.4.2 Bounded Property Checking . . . 19
  - 2.4.3 Formal Proof for Unbounded Model . . . 20
- 2.5 Implementation and Evaluation . . . 22
  - 2.5.1 Experiment Setup . . . 22
  - 2.5.2 Experiment Results . . . 24
  - 2.5.3 Threats to Validity . . . 30
- 2.6 Related Work . . . 30
  - 2.6.1 Smart Contract Analysis and Verification . . . 31
  - 2.6.2 Fairness Checking in Software Systems . . . 32
  - 2.6.3 Mechanism Design and Game Theory . . . 33
- 2.7 Conclusion . . . 34

3 Testing **35**
3.1 Model-Based Testing 35
3.1.1 ModCon Overview 38
3.1.1.1 User Interface 38
3.1.1.2 Back-End Implementation 41
3.1.2 Evaluation 42
3.1.3 Related Work 44
3.1.4 Conclusion 44
3.2 Semantic Test Oracle of Effective Fuzzing 45

4 Access Control Analysis **47**
4.1 Introduction 47
4.2 Motivating Example 49
4.3 User Permission Model 50
4.3.1 Role-Based Access Control Model 51
4.3.2 Security Policies on Top of User Permissions 52
4.4 Framework 54
4.4.1 The User Role Mining 54
4.4.2 SFSM Learning 56
4.4.3 Security Policy Inference 57
4.4.4 Testing 58
4.4.5 Abstraction and Model Checking 59

5 Conclusion and Future Work **61**
5.1 Future Work 61

Bibliography **63**

List of Figures

1.1 The smart contract transaction on Ethereum. 2
1.2 The thesis overview of the current work and future work. 5

2.1 The CryptoRomeAuction Solidity source code. 10
2.2 The mechanism model of CryptoRomeAuction with three bidders. . 11
2.3 Syntax of the auction/voting mechanism model. 15
2.4 Semantic rules of the auction/voting model. 16
2.5 Workflow of the FairCon framework. 16
2.6 The harness program for mechanism model orchestration. 18
2.7 The EtherAuction Solidity source code. 26
2.8 Model extraction time with increasing number of bidders. 27
2.9 Property checking time with increasing number of bidders. 28

3.1 Illustration of a CMA smart contract at WeBank. 36
3.2 Architecture of ModCon. 38
3.3 Smart contract deployment and setup. 39
3.4 User-configured model specification. 40
3.5 Test generation control panel. 40
3.6 State and transition coverage achieved for CMA and `BlindAuction`. 43

4.1 Illustration of the time-travel-based security policy validation. . . . 48
4.2 The abstracted Solidity contract of Dicether. 49
4.3 Overview of security policy. 52
4.4 Execution tree of Dicether 53
4.5 Workflow of the SpCon Framework. 54
4.6 The extraction of user accesses. 55
4.7 The roles of Dicether. 56
4.8 The SFSM of Dicether. 57
4.9 The information flow between roles of Dicether. 58
4.10 Smart contract testing 59
4.11 The control flow graph of Dicether's *userCancelGame*. 59

List of Tables

2.1 Example instances of CryptoRomeAuction. 9
2.2 Fairness checking on auction contracts. 24
2.3 Fairness proving on fair auction contracts in Table 2.2. 24
2.4 Fairness checking on voting contracts. 25

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 Smart Contract

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. A *blockchain* is often called a distributed ledger recording transactions, consisting of blocks linked by cryptographic hashes and managed by a peer-to-peer network. *Ethereum* is one of the most popular blockchains today, as it supports smart contracts. A smart contract transaction executed on Ethereum can transition the blockchain from a state to another state. Figure 1.1 briefly shows smart contract transaction on Ethereum. A user sends a transaction to a miner for executing a specific contract function at block n. The miner then reads the blockchain state database and runs on it the called function whose operation codes are interpreted by Ethereum virtual machine (EVM). Later, the blockchain state database transitions and the miner will commit the new state database to a mined block that is appended as block $n + 1$ after validation. Finally, the user gets a receipt to confirm the transaction result which includes transaction status, event logs, ether transferred and gas consumption.[1] Particularly, if transaction status indicates success, the contract state will transition to new one as well. Otherwise, it will remain unchanged.

Meanwhile, most smart contracts are designed as an organization for a social purpose, where user explicitly or implicitly play a role when they participate in

[1]Gas is the fee paid to the miner by transaction sender.

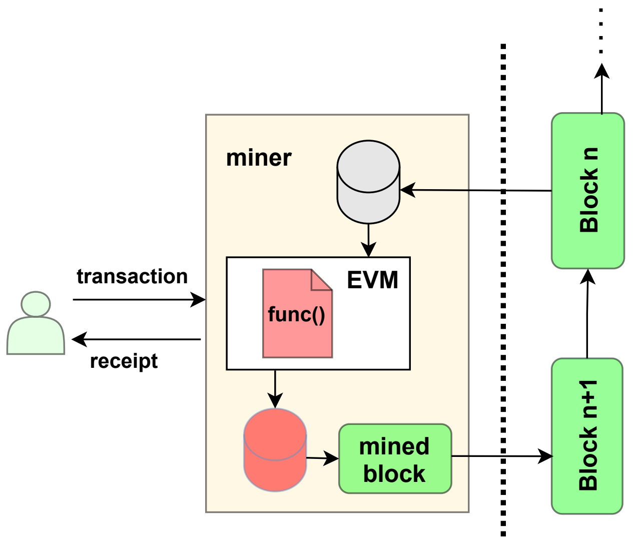


FIGURE 1.1: The smart contract transaction on Ethereum.

smart contract. These smart contracts have been widely adopted in many areas including auctions, finance, gambling, exchanges, governance and etc.. For instance, the Ethereum Name Service (ENS) is one of the decentralized applications (DApp) that uses auction smart contract to provide service for the bids on Ethereum domains; and a gambling DApp, i.e., Dicether uses contracts to maintain houses and users' stake in the games; while CryptoKitties uses contracts to organize a blockchain exchange market for the selling and breeding of digital pets implemented as non-fungible tokens (NFTs). With smart contracts managing larger sum of valuable assets, it is critical to ensure smart contracts achieve the security and reliability.

1.2 Motivation and Challenge

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 [1], 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* [2] 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, current works mainly rely on the priori security patterns, e.g., reentrancy, delegatecall, gasless send and tainted owner. Although these priori security patterns has been proven effective in finding smart contract vulnerabilities, they are very limited in finding deeper smart contract vulnerabilities such as liquidity bug, data integrity bug and etc.

However, for most smart contracts on permissionless blockchains, system behaviours and requirements are not specified, thus making it hard to ensure security of smart contracts. Even though, popular Ethereum DApps have attracted many users in thousands or even millions of transactions [3] that occurred in the past. Likely security policies such as information integrity can be learned via time travel on transaction history of smart contracts. With likely security policies, it is easy to find security bugs. Particularly, a runtime verification for dynamic security policy can also be integrated with blockchain environment for ensuring smart contract security.

The reliability of smart contracts also matters. Little work has been done on 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 computational complexity of contracts (e.g., up to \$15 in fees [4]). 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 [5]. The smart contracts powering these enterprise applications are more sophisticated and often demonstrate strong stateful behaviors. However, existing testing and analysis tools [6–10] target only simple smart contracts and mainly focus on their security issues, which will not work well in these stateful enterprise smart contracts due to the lack of understanding of system behaviours, absence of test oracles and missing measurement of test adequacy. The aforementioned enterprise smart contracts on permissioned blockchains are usually well specified, which make it suitable for model-based testing (MBT).

Fairness is also a critical factor of smart contract reliability. However, fairness issues of smart contracts have not yet attracted much attention. Smart contract 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" [11] 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 [12]. 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.

1.3 Research Objective

In summary, the research objective is to address the aforementioned challenges on security and reliability of smart contracts. In a word, these challenges include the lack of specification, model-based testing and general fairness analysis framework mainly for reliability and effective fuzzing, access control analysis and runtime verification mainly for security.

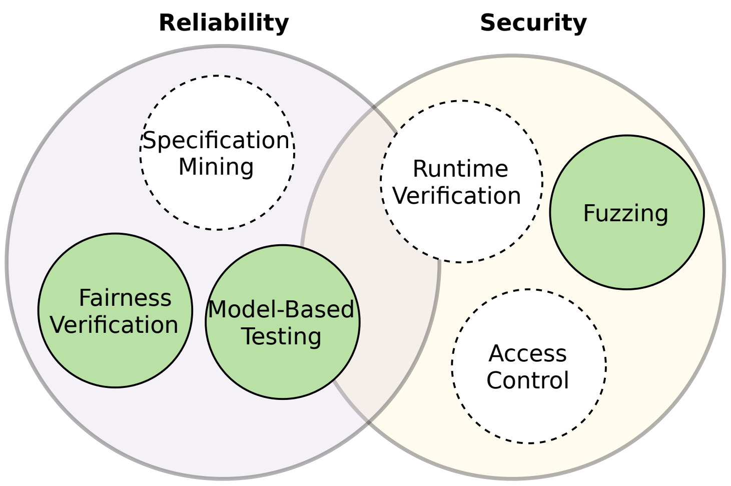


FIGURE 1.2: The thesis overview of the current work and future work.

Figure 1.2 outlines the current and future research topics for achieving enforcing security and reliability of smart contracts. Chapter 2 will propose a general fairness verification framework for smart contracts. The model-based testing frameworks for smart contracts will be presented in Chapter 3 followed by the semantic test oracle of effective fuzzing. The access control analysis framework will be discussed in Chapter 4, while the future work will be briefed in Chapter **??**. Finally this report will be concluded in Chapter 5.

1.4 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. 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/OBEVRT`.

- 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 to be satisfied for each transition, invariants, and the mapping from the model to the contract code. With the test model given, users can further customize the testing process by choosing from different coverage strategies and test prioritization options. ModCon then generates tests with the goal of exercising as many system behaviors as possible while prioritizing on cases of particular interests. Any violation of the specified oracle is recorded and reported to users. ModCon has a Web-based interface, providing easy access to all the testing capabilities and customization options. Source code and a video demonstrating the usage of ModCon are available at https://sites.google.com/view/modcon.

Chapter 2

Fairness Verification

In this chapter, we present FairCon, a framework for verifying fairness properties of smart contracts.

2.1 Introduction

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 [13, 14]), 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* [15] 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` [3]. 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.

Organizations. The rest of this chapter 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. Section 2.5 gives details on the implementation and presents the evaluation results. Sections 2.6 and 2.7 compare FairCon with the related work and summarize the chapter, 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.

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

Example 2.1. Figure 2.1 shows a simplified Ethereum smart contract, named `CryptoRomeAuction`, written in Solidity [16], 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.

Threats to Contract Fairness. One way that `CryptoRomeAuction` can become unfair to the participants is through the so called *shill bidding* [17]—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 2.1 to demonstrate. Suppose there are three bidders, p_1, p_2, and p_3, participating in the

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

```
contract CryptoRomeAuction {
  /** FairCon Annocations
   @individual(msg.sender, msg.value, VALUE)
   @allocate(highestBidder)
   @price(highestBid)
   @outcome(bid())
  */
  uint256 public highestBid = 0;
  address payable public highestBidder;
  mapping(address=>uint) refunds;
  function bid() public payable{
    uint duration = 1;
    if (msg.value < (highestBid + duration)){
      revert();
    }
    if (highestBid != 0) {
      refunds[highestBidder] += highestBid;
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
}
```

Figure 2.1: The CryptoRomeAuction Solidity source code.

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.

$$
\begin{aligned}
&\texttt{CryptoRomeAuction} := (\mathit{msgsender}_1, \mathit{msgvalue}_1, \_) \\
&\quad (\mathit{msgsender}_2, \mathit{msgvalue}_2, \_) \\
&\quad (\mathit{msgsender}_3, \mathit{msgvalue}_3, \_) \\
&\quad \texttt{assume}:\ (\texttt{not}\ (\mathit{msgvalue}_2 < \mathit{msgvalue}_1 + 1))\ \texttt{and} \\
&\qquad (\texttt{not}\ (\mathit{msgvalue}_3 < \mathit{msgvalue}_2 + 1)) \\
&\quad \texttt{allocate}:\ \texttt{argmax}(\mathit{msgvalue}_1, \mathit{msgvalue}_2, \mathit{msgvalue}_3) \\
&\quad \texttt{price}:\ \texttt{max}(\mathit{msgvalue}_1, \mathit{msgvalue}_2, \mathit{msgvalue}_3)
\end{aligned}
$$

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

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.

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 [15] 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 faircon, such as Dafny [18], to perform the automated verification.

2.3 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 [13, 14]. 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., $\forall \hat{\theta} \in \Theta \forall t' \cdot \sum_i t_i(\hat{\theta}) \geq \sum_i t'_i(\hat{\theta})$.

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., $\{\hat{\theta}_1, \ldots, \hat{\theta}_{i-1}, \hat{\theta}_{i+1}, \ldots, \hat{\theta}_{j-1}, \hat{\theta}_{j+1}, \ldots, \hat{\theta}_n\}$.

$$
\begin{aligned}
\texttt{<individual>} &:= (\mathit{id} : \mathbb{S}, \mathit{bid} : \mathbb{N}, \mathit{val} : \mathbb{N}) \\
\texttt{<func>} &:= \texttt{max} \mid \texttt{argmax} \\
\texttt{<exp>} &:= \texttt{<individual>}.\mathit{id} \mid \texttt{<individual>}.\mathit{bid} \\
&\quad \mid \mathbb{N} \mid \texttt{<exp> [+-] <exp>} \mid \texttt{<func>(<exp>*)} \\
\texttt{<bool>} &:= \texttt{<exp> == <exp>} \mid \texttt{<exp> < <exp>} \\
&\quad \mid \texttt{not <bool>} \mid \texttt{<bool> and <bool>} \\
\texttt{<assumption>} &:= \texttt{assume} : \texttt{<bool>} \\
\texttt{<outcome>} &:= \texttt{allocate} : \texttt{<exp>} \\
&\quad \mid \texttt{price} : \texttt{<exp>};\ \texttt{allocate} : \texttt{<exp>} \\
\texttt{<property>} &:= \texttt{<bool>} \mid \texttt{forall} : \texttt{<bool>} \\
\texttt{<mechanism>} &:= \texttt{<individual>*};\ \texttt{<assumption>};\ \texttt{<outcome>}; \\
&\quad \texttt{<property>}
\end{aligned}
$$

FIGURE 2.3: Syntax of the auction/voting mechanism model.

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.

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

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

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

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

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

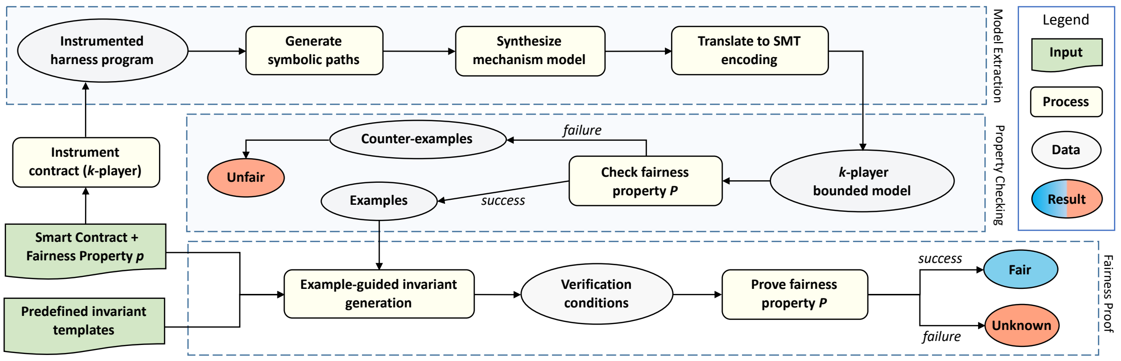

Model Extraction
Instrumented harness program
Generate symbolic paths
Synthesize mechanism model
Translate to SMT encoding
Instrument contract (k-player)
Smart Contract + Fairness Property p
Predefined invariant templates
Property Checking
Counter-examples
failure
Unfair
Check fairness property P
k-player bounded model
Examples
success
Fairness Proof
Example-guided invariant generation
Verification conditions
Prove fairness property P
success
Fair
Unknown
failure
Legend
Input
Process
Data
Result

FIGURE 2.5: Workflow of the FAIRCON framework.

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. 4.6. 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) [19] and ERC-1815 (blind auctions, under review) [20] 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

In this section, we present the FairCon verification framework for smart contract fairness. Figure 4.5 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

```
 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 2.6: The harness program for mechanism model orchestration.

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,

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

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. 4.5). The followings are three common types of invariants required for auctions.

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

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

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

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

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

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* [21] 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, $sp(x = 2,$ ``` ``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 [18] 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. 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.

2.5 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. 2.4.1), and then automatically perform symbolic path analysis on the model (c.f. Sect. 2.4.2). 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 [18] is used to check the two validity conditions of invariants (c.f. Sect. 2.4.3) 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?

2.5.1 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 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

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

TABLE 2.2: 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 2.3: Fairness proving on fair auction contracts in Table 2.2.

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

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/OBEVRT`.

2.5.2 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. 2.5.1. Tables 2.2 and 2.4 show the results for fairness checking on auction and voting contracts, respectively.

Table 2.4: 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

In Table 2.2, 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), 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 2.3 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 2.2. 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. 2.7, 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`,

```
 1 contract EtherAuction {
 2   //Anyone can bid by calling this function and supplying the
   ↪  corresponding eth
 3   function bid() public payable {
 4     require(auctionStarted);
 5     require(now < auctionEndTime);
 6     require(msg.sender != auctioneer);
 7     // If sender is already the highest bidder, reject it.
 8     require(highestBidder != msg.sender);
 9     address _newBidder = msg.sender;
10     uint previousBid = balances[_newBidder];
11     uint _newBid = msg.value + previousBid;
12     // Each bid has to be 0.05 eth higher
13     if (_newBid  == highestBid + (5 * 10 ** 16)) return;
14     // The highest bidder is now the second highest bidder
15     secondHighestBid = highestBid;
16     secondHighestBidder = highestBidder;
17     highestBid = _newBid;
18     highestBidder = _newBidder;
19     latestBidTime = now;
20     // Update the bidder's balance so they can later withdraw
   ↪  any pending balance
21     balances[_newBidder] = _newBid;
22   }
23 }
```

FIGURE 2.7: The EtherAuction Solidity source code.

`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 2.4 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 Table 2.4, because the diverse $\mathbb{R}$ valuation of proposals brings the incentive for voters to lie and to conspire with

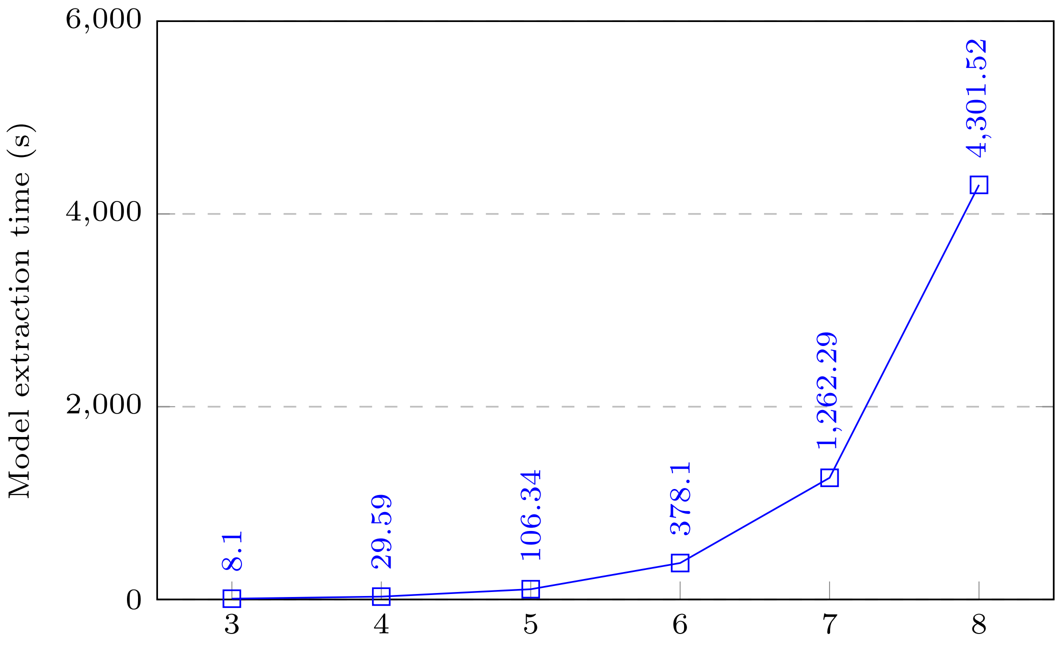


Figure 2.8: Model extraction time with increasing number of bidders.

others. In the $\{0,1\}$ valuation setting, as shows in third large column of Table 2.4, 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 2.4, 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 2.4 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. 2.3.2). 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 2.2 and 2.4.

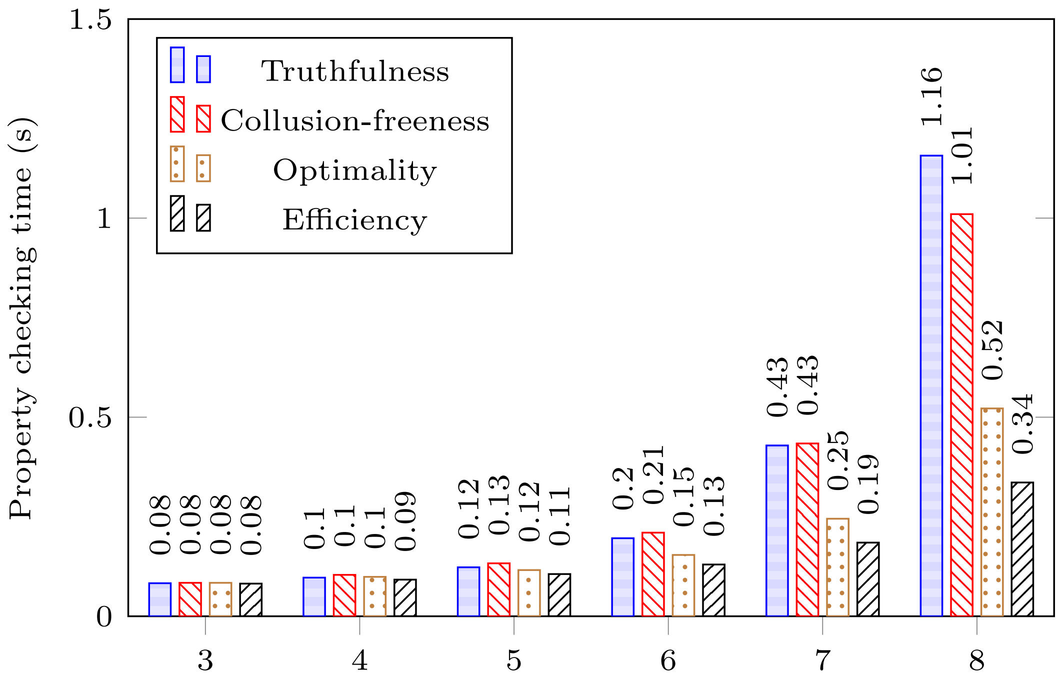


FIGURE 2.9: Property checking time with increasing number of bidders.

To explore the efficiency of FAIRCON further, we selected the `CryptoRomeAuction` contract to make performance experiments on FAIRCON. Figures 2.8 and 2.9 show the execution time for mechanism model extraction and fairness property checking when the number of bidders increases, respectively. In Fig. 2.8, 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 2.9 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. 2.8, 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 2.2 and $k = 5$ in Table 2.4). Otherwise, as shown in Table 2.3, 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. 2.7) 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.

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

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.

2.5.3 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.

2.6 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.

2.6.1 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 [22]. For example, a smart contract which fails to check the return value of a (possibly failed) external call operation, has the *unchecked call* vulnerability [23, 24]. The execution logic of a smart contract that is not independent of environmental variables, e.g., the block timestamp, is prone to *dependence manipulation* [25], including the *timestamp dependency* vulnerability [26]. 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 [27]. Absence of proper checks for arithmetic correctness make Ethereum contracts prone to *integer and batch overflow/underflow* [28, 29]. Furthermore, the progress of a smart contract can be compromised by gas-exhaustive code patterns [30, 31].

To address these security issues, Ellul et al. developed a runtime verification technique, ContractLarva [32], to rule out certain unsafe behaviors during the execution of smart contracts. Oyente [7, 26] is one of the first to detect smart contract vulnerabilities using symbolic execution. ContractFuzzer [6] is among the early fuzz testing tools and Mythril [33] 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 [8–10, 34–37] 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^* [38], 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 [23]. Hildenbrandt et al. presented an executable formal semantics for the Ethereum platform, named KEVM [39], based on which, Park et al. [40] 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 [41, 42]. Abdellatif et al. [43] formalized blockchain and users' behaviors to verify properties about their interactions using statistical model checking. Nehai et al. [44] 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.

2.6.2 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 [45–47], either learned from data or created by experts. The term "fairness" can be subjective depending on the actual contexts. Verma and Rubin [48] 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. [47] 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. [49] 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. [**?**] found through a survey that nearly 0.05% of the transactions on Ethereum could be owing to Ponzi schemes.

Chen et al. [50] 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.

2.6.3 Mechanism Design and Game Theory

Mechanism design has been well studied in the economic domain [13, 51–53]. 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 [51] 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 [13] presented mechanism theory in a full view and provided formal definitions to many concepts belonging to this domain, while Klemperer [52] introduced the most fundamental concepts for auction and carried out a thorough analysis of optimal auctions, the equivalence theorem, and marginal revenues. Lehmann [53] 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. [54] implemented a Vickrey second price auction on a smart contract to setup and operate a market of energy exchanges. Similarly, Chen et al. [55] provided an e-auction mechanism based on blockchain to ensure confidentiality, non-repudiation, and unchangeability of the electronic sealed bid. CReams [12] implemented a collusion-resistant k-Vickery auction. Galal and Youssef [56] presented a smart contract protocol for a succinctly verifiable sealed-bid auction on the Ethereum blockchain to protect bidders' privacy. Mccorry et al. [57] proposed the first implementation of a decentralized and self-tallying internet voting protocol using smart contract to guarantee secure e-voting. Bigi et al. [58] 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. [59] 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.

2.7 Conclusion

In this chapter, 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.

Chapter 3

Testing

In this chapter, we will present a model-based testing framework for smart contracts as well as a semantic test oracle of effective fuzzing.

3.1 Model-Based Testing

The existing blockchain networks can be broadly categorized into the permissionless and permissioned blockchains, where the former is open to the public (e.g., Bitcoin [60] and Ethereum [61]) and the latter is only accessible to trusted private groups or individuals (e.g., Hyperledger Fabric [62]). The consortium/federated blockchains (e.g., FISCO BCOS [5] and Azure Blockchain Workbench [63]) sit somewhere in the middle: they are suitable for use between multiple businesses or organizations for performing transactions and exchanging information. One major difference between smart contracts on the permissioned and permissionless blockchains is that the contract execution on permissionless chains is bounded by resource constraints. For example, on Ethereum, one has to pay miners a certain amount of "gas" (cryptocurrency on Ethereum) as the transaction fee to deploy or call contract, which is largely decided by the complexity of the contract (e.g., up to \$15 in fees [4]). Therefore, to reduce the gas consumption, smart contracts on permissionless chains are often kept simple, making it unsuitable for implementing enterprise applications with complex business logic.

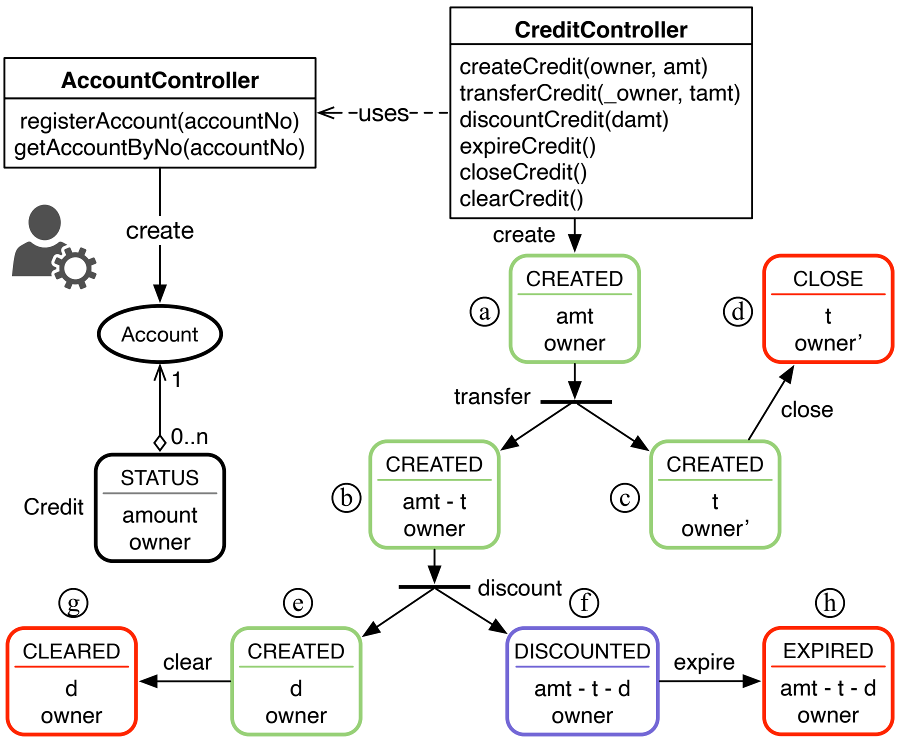


FIGURE 3.1: Illustration of a CMA smart contract at WeBank.

At the same time, smart contracts have been used to implement many industrial applications of high complexity and production quality on permissioned and consortium blockchains. Unlike the permissionless blockchains, such as Bitcoin and Ethereum mainly used for cryptocurrency exchange (e.g., ERC Token and DeFi applications), 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 [5]. The smart contracts powering these applications are more sophisticated and often demonstrate strong stateful behaviors.

Example. Figure 3.1 illustrates some usage scenarios of a Credit Management Application (CMA) at WeBank [64], implemented using smart contracts, running on FISCO BCOS consortium blockchain. CMA is used to handle inventory and asset management in supply chain through a blockchain-based credit system, which can facilitate credit transfer among different business owners and help small businesses receive instant financial support securely.

The user first deploys an `AccountController` contract, whose address is then used to instantiate the `CreditController` contract. `AccountController` is in charge of the account creation and management. An account may own `Credit`(s), which are transferable and divisible tokens with stipulated values. The state of a `Credit` is captured by the tuple, $(STATUS, amount, owner)$, whose fields represent the status, value captured, and its ownership, respectively. A `Credit` instance supports credit operations including creation, transfer, discount, expiration, clearance, and closure. Through `CreditController`, one can first create a credit, namely, ⓐ, under the specified `Account`. In this case, a `transfer` operation is executed on ⓐ, thus dividing ⓐ into two new credits, namely, ⓑ and ⓒ. By design, the total value of ⓑ and ⓒ equals to that of ⓐ. Then a `discount` operation is applied on ⓑ, resulting in a newly created credit ⓔ and a discounted credit ⓕ. By design, the total value of ⓔ and ⓕ equals to that of ⓑ, but the status of ⓕ becomes "DISCOUNTED". To complete the life cycle of a credit, one may apply either the `close`, `clear`, or `expire` operation, bringing the credit into the "CLOSED" (e.g., ⓓ), "CLEARED" (e.g., ⓖ), or "EXPIRED" (e.g., ⓗ) state, respectively. Once a credit is in "CLOSED/CLEARED/EXPIRED", it should no longer accept further operation.

Existing testing and analysis tools target Ethereum smart contracts and mainly focus on their security issues. Such tools do not work well on this example for the following reasons. **(1) Lack understanding of system behaviors**. The different states of a credit instance is implemented with special encoding. For example, the $STATUS$ field is encoded as bit-vectors for performance considerations. It is unclear how to interpret system states and behaviors at these states without this knowledge. **(2) Absence of oracle**. Existing tools may rely on implicit security properties (e.g., underflow/overflow and exceptions) as oracle, which is absent when the functional correctness is concerned. The expected system behavior (e.g., "EXPIRED" is terminal) is not known prior and should be provided by the contract designer. **(3) Missing measurement of test adequacy**. The traditional coverage criteria used by existing tools, such as branch and path coverage, are not good measurement of test adequacy for this example. Covering every single path of the contract program does not equal exercising all system states and state transitions. It is challenging to navigate through all system behaviors without proper adequacy measurements.

ModCon. To address these challenges, we propose ModCon, a model-based testing platform for smart contracts. ModCon targets enterprise smart contract applications written in Solidity [16] from permissioned/consortium blockchains such as FISCO BCOS, but is also compatible with Ethereum.

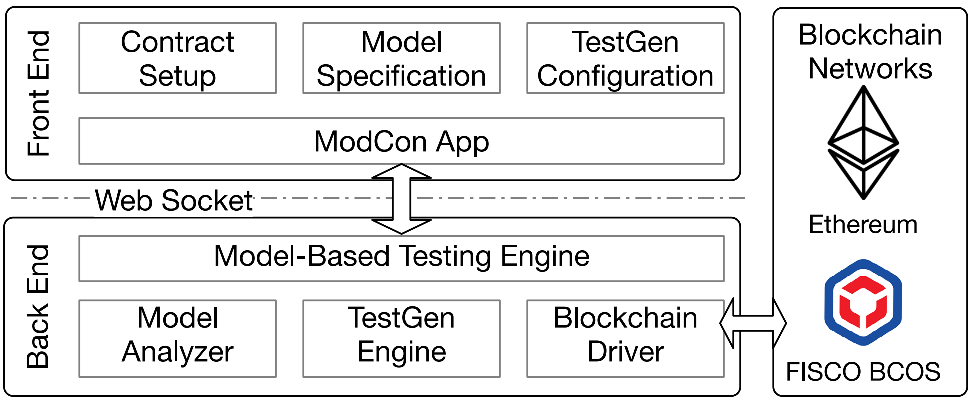


FIGURE 3.2: Architecture of MODCON.

MODCON allows users to specify system models and define test oracles, which are then used to guide the test generation and execution. The key features of MODCON include test-model specification, customized test generation and user friendly web-based interface.

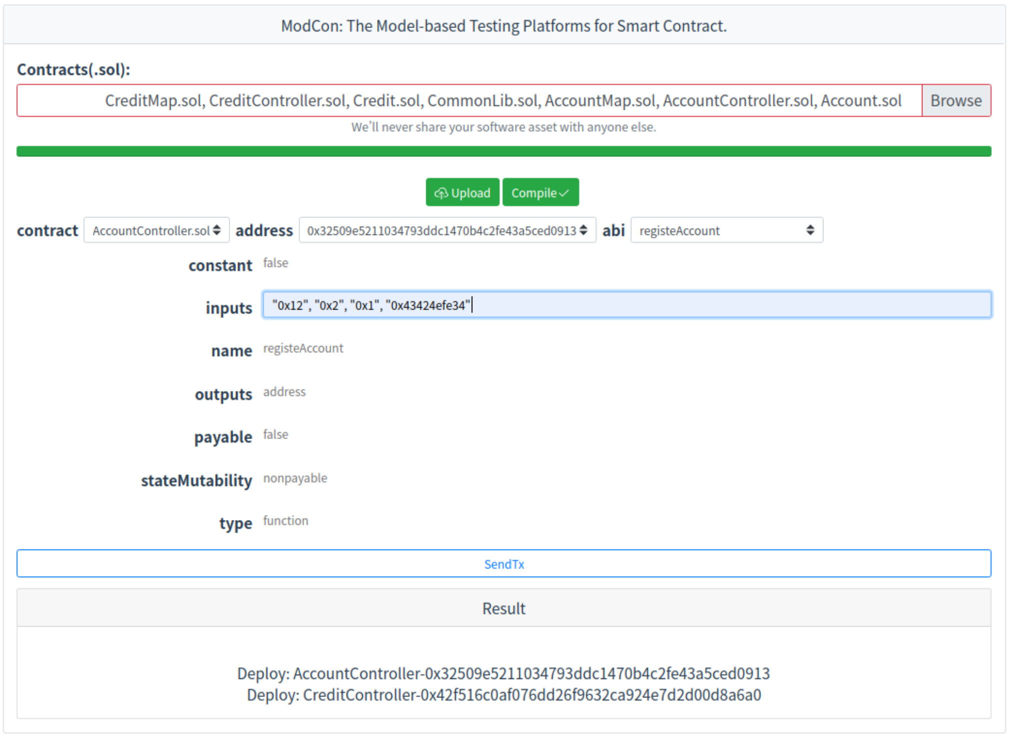
3.1.1 ModCon Overview

In this section, we describe the architecture of MODCON and demonstrate its user interface. As shown in Fig. 3.2, MODCON consists of a web-based front end (implemented as a Vue.js [65] application) and a server-side back end (implemented on top of the Node.js JavaScript runtime [66]). The front-end accepts two inputs from users: the target smart contracts and the test-model specifications to drive the model-based testing process. The front-end allows users to specify coverage strategies and configure test generation priority, and the test execution progress can be monitored on-the-fly. The back-end communicates with the front-end through the WebSocket. On the back-end, the model-based testing engine is in charge of smart contract compilation/deployment, model specification analysis, and the customized model-based testing tasks as per users' requests.

3.1.1.1 User Interface

The user interface of MODCON mainly supports three tasks, namely, contract setup, model specification, and testing controls.

Contract Setup. First, users are to upload all relevant smart contract source files, which are then automatically compiled and deployed onto the blockchain network.


ModCon: The Model-based Testing Platforms for Smart Contract.

Contracts(.sol):

CreditMap.sol, CreditController.sol, Credit.sol, CommonLib.sol, AccountMap.sol, AccountController.sol, Account.sol | Browse

We'll never share your software asset with anyone else.

contract AccountController.sol **address** 0x32509e5211034793ddc1470b4c2fe43a5ced0913 **abi** registeAccount

constant	false
inputs	"0x12", "0x2", "0x1", "0x43424efe34"
name	registeAccount
outputs	address
payable	false
stateMutability	nonpayable
type	function

Result

```
Deploy: AccountController-0x32509e5211034793ddc1470b4c2fe43a5ced0913
Deploy: CreditController-0x42f516c0af076dd26f9632ca924e7d2d00d8a6a0
```

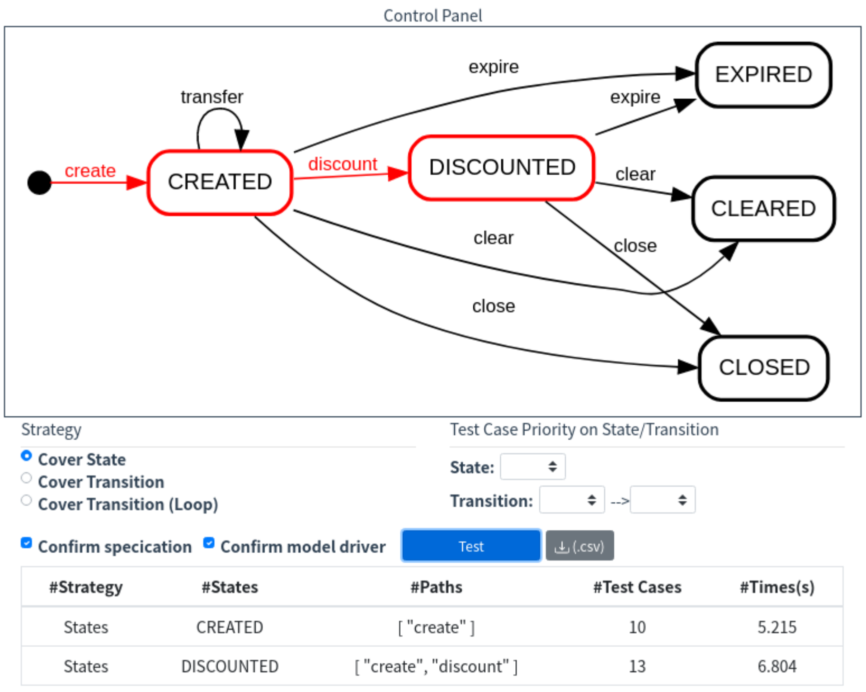
FIGURE 3.3: Smart contract deployment and setup.

Once the contracts are successfully deployed, users can directly interact with them by sending transactions, and the transaction receipts are displayed on the result pane below. For example, as shown in Fig. 3.3, seven contracts related to the CMA application (i.e., `Account`, `AccountController`, `Credit`, `CreditController`, ...) had been uploaded to ModCon. `AccountedController` and `CreditController` were deployed at addresses, "`0x325...913`" and "`0x42f...6a0`", respectively. One transaction, calling the `registerAccount` function, was sent to `AccountController` to create an account which would be used to hold credit instances. The target contracts' information, including the ABIs and deployment details, are cached and will be used for test case generation in a later stage.

Model Specification. Figure 3.4 shows an abridged test-model specification for the CMA application, which user can customize for his/her applications. The "`id`" and "`main`" fields indicate the model identifier and the entry contract, respectively. The "`contracts`" field lists all relevant contract dependencies required in the test model. The "`states`" and "`transitions`" fields jointly define a state machine model for the target application. The "`actions`" field establish a mapping between functions from the contract implementation and the actions that can be taken to perform state transitions.

```
"id": "CMA#1",
"main": "CreditController",
"contracts": {
  "CreditController": {"address": "0x00", "name": "CreditController"}
},
"actions": {
  "create": {"CreditController": ["createCredit"]},
  "discount": {"CreditController": ["discountCredit"]},
  "transfer": {"CreditController": ["transferCredit"]},
  "expire": {"CreditController": ["expireCredit"]},
  "clear": {"CreditController": ["clearCredit"]},
  "close": {"CreditController": ["closeCredit"]}
},
"states": [
  { "name": "CREATED", "type": "regular", "Predicate": "status[0] == CREATED" },
  ↪  ...
],
"transitions": [
  { "from": "INITIAL", "to": "CREATED", "action": "create" },...
]
```

FIGURE 3.4: User-configured model specification.



Strategy

- [✓ Cover State]
- Cover Transition
- Cover Transition (Loop)

Test Case Priority on State/Transition

State:

Transition: -->

[✓ Confirm specication] [✓ Confirm model driver]

#Strategy	#States	#Paths	#Test Cases	#Times(s)
States	CREATED	["create"]	10	5.215
States	DISCOUNTED	["create", "discount"]	13	6.804

FIGURE 3.5: Test generation control panel.

TestGen Configuration. The test-model specification (i.e., Fig. 3.4) provided by users is visualized as a state machine diagram shown in Fig. 3.5. Users may further customize the test generation process by choosing from the three coverage strategies: (1) *cover states*, aiming to cover every states, (2) *cover transitions*, aiming to cover every transitions, and (3) *cover transitions (loop)*, aiming to cover every transitions including loops. Based on experiments, covering loop transitions may increase testing costs without covering new states, but it can help discover corner cases and verify the integrity of the test-model. In addition, users may prioritize the test generation leaning towards specific states or transitions. As shows in Fig. 3.5, the “cover states” strategy is selected and the states `CREATED` and `DISCOUNTED` are covered by 10 and 13 test cases, respectively.

3.1.1.2 Back-End Implementation

The model-based testing engine consists of three parts: i.e., model analyzer, test generation engine, and blockchain driver.

Model Analyzer. The model analyzer reads the model specification from the front-end and automatically translates it into a test driver written in JavaScript. The test driver stipulates how tests should be generated and executed, which is then displayed in the front-end client for users’ confirmation and customization. For instance, users may insert additional test oracles in the form of pre/post conditions and assertions.

TestGen Engine. The test generation (TestGen) engine receives testing requests and collects the test-model related information from the front-end, which includes the confirmed test driver, the coverage strategies, and the test generation priorities. The engine first computes all logical transition paths following the specific coverage strategies and goals using graph searching algorithms. For example, to reach the `CLEARED` state of CMA shown in Fig. 3.5, the logical transition paths for different strategies are listed below.

- Cover states: INITIAL → CREATED → CLEARED.
- Cover transitions: INITIAL → CREATED → CLEARED; INITIAL → CREATED → DISCOUNTED → CLEARED.
- Cover transitions (loop): INITIAL → CREATED → CREATED → CLEARED; INITIAL → CREATED → CREATED → DISCOUNTED → CLEARED.

The TestGen engine ranks these logical transition paths based on the order defined by the test case priorities, and then generates concrete test cases (with concrete input values and environment settings) corresponding to each logical transition path. The generation of concrete input values adopts standard techniques, such as the mutation-based method in ContraMaster [10, 67], with seed pools for different input types. Built upon the blockchain driver, the TestGen engine sends these concrete test cases to blockchain platforms for execution and monitors the execution status at the same time. The engine keeps generating test cases for execution until the maximum time budget or failure limit is reached. During test execution, the engine reports the testing results back to the front-end client, which displays the current progress in real-time.

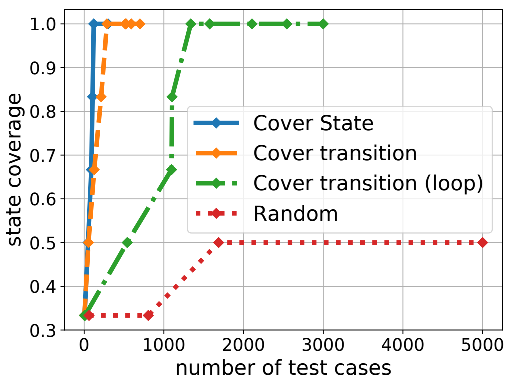
Blockchain Driver. The blockchain driver directly interacts with the blockchain networks for contract deployment and establishes a transaction interface with the networks. Currently, ModCon supports two blockchain platforms, namely, Ethereum and FISCO BCOS. It can easily be extended to other blockchain platforms.

3.1.2 Evaluation

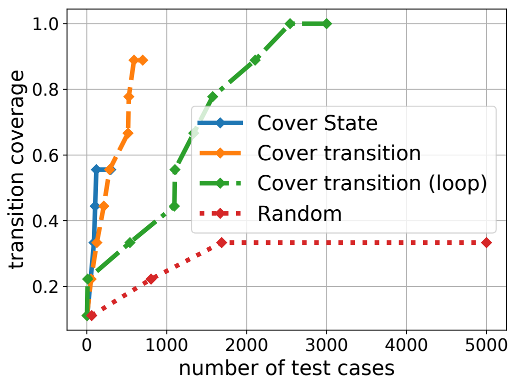
In this section, we evaluate ModCon on the CMA smart contract application from WeBank and the `BlindAuction` contract used by FSolidM [68], a state machine based smart contract code generator.

We manually constructed their model specifications with the help from the contract developers and the related documentation. The experiments were conducted on a desktop computer with Ubuntu 18.10 OS, an Intel Core i5 2.50 GHz processor and 8GB RAM. All cases were evaluated on the FISCO BCOS blockchain.

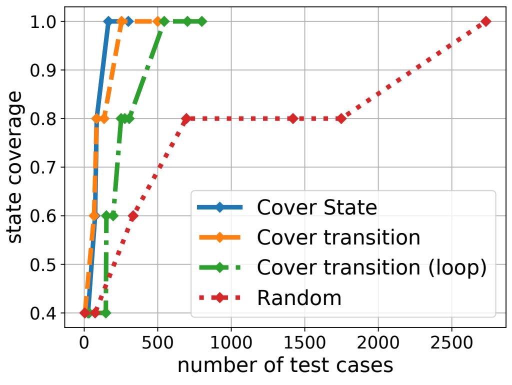
Figure 3.6 shows the evaluation results. The vertical and horizontal axes represent the state/transition coverage and the number of test cases, respectively. We examined aforementioned three coverage strategies and compared the results of ModCon with random testing. Among these strategies, the results show that the cover state strategy first reaches all states of both CMA and `BlindAuction`, while the strategy to cover transition including loops has the potential to reach all states and explore more transitions at the cost of more test cases. All of the three proposed strategies achieve much higher state and transition coverage than random testing, which shows that random testing is not suitable to deal with enterprise smart contract applications. Random testing achieves lower state and transition coverage in CMA than those in `BlindAuction`, because the former is of higher complexity in its business logic and state encoding than the latter. For example, CMA uses a bit-vector of more than 11 bits as its function input or to encode the *STATUS*, and blindly enumerating bit-vector values is extremely inefficient.



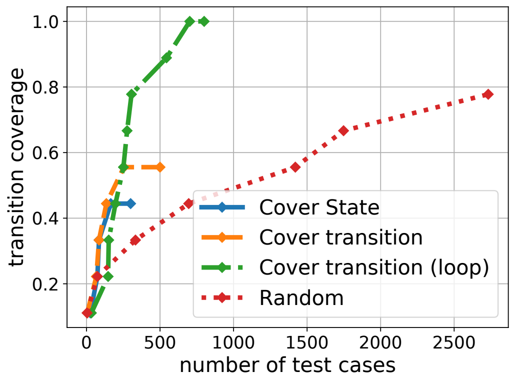
(A) CMA: state cov.



(B) CMA: transition cov.



(C) `BlindAuction`: state cov.



(D) `BlindAuction`: transition cov.

FIGURE 3.6: State and transition coverage achieved for CMA and `BlindAuction`.

In our experiments, MODCON was able to reach all states and transitions for each case within about 500 test cases. This is mainly because of the guidance from the test-model, which makes MODCON effective on enterprise smart contract applications such as CMA. Additionally, with the test-model specification, MODCON allows users to define test oracles in the generated test driver. For example, the specification of CMA requires `CLOSED`, `CLEARED`, and `EXPIRED` to be final states, which means no transition shall be made once the system falls into one of the three states. We insert this specification as a test oracle into the test driver and discovered violations against it in the original implementation of CMA. The transitions between `EXPIRED`, `CLOSED`, and `CLEARED` were possible due to an implementation error. We reported this error to the CMA developer team from WeBank, and they confirmed it to be a real bug. The demonstration video of MODCON, along with more cases and experiment results, can be accessed at: https://sites.google.com/view/modcon.

3.1.3 Related Work

Most of the existing testing and analysis tools focus on the security issues of Ethereum smart contracts. Oyente [7, 26] is one of the first static analyzer detecting security vulnerabilities in smart contracts based on symbolic execution. It searches for violations of predefined security properties without actually executing the contract program. Other notable static security analysis tools include Zeus [34], Mythril [33], sCompile [36], and Securify [35]. In contrast, the dynamic tools instrument either the contract code or the Ethereum Virtual Machine (EVM) and observe anomalies during runtime execution. ContractFuzzer [6] is the earliest dynamic fuzz testing tool aiming a number of common vulnerability types, including the reentrancy, exception disorder, block dependency, etc. Other fuzzing tools follow similar principles: e.g., Reguard [69], ContraMaster [10, 67], and sFuzz [70]. These tools are not designed for testing functional correctness, and as mentioned in Sect. 3.1, they are not suitable for enterprise smart contract applications either.

There are several recent works on the functional correctness of Ethereum smart contracts. VeriSol [71] relies on formal verification to check the semantic conformance between a contract implementation and its workflow policy. The policy is provided by users, describing the high-level workflow of the application in a style similar to our model specifications. FSolidM [68] and VeriSolid [72] both aim to facilitate the creation of correct-by-design contracts, with emphases on the security and functional aspects, respectively, where a finite state machine is used as the contract specification to capture the expected system behaviors. ModCon is based on the idea of model-based testing [73], which uses an explicit abstract model of the target contract to automatically derive tests. It serves as a complement to other static validation/construction techniques in providing more flexible and accurate quality assurance solutions.

3.1.4 Conclusion

In this section, we described the architecture of ModCon, its user interface, and prominent features. We also demonstrated the effectiveness of it on real smart contract applications from WeBank. The model-based testing capability of ModCon enables it to generate higher-quality test cases for enterprise smart contracts from permissioned and consortium blockchains.

3.2 Semantic Test Oracle of Effective Fuzzing

Chapter 4

Access Control Analysis

4.1 Introduction

Smart contracts deployed on permissionless blockchains, such as Ethereum, are accessible to any user in a trust-less environment. Therefore, most smart contract applications implement access control policies to protect their valuable assets from unauthorized accesses. The correctness of such policies is critical for maintaining smart contract security. For example, suicidal contracts [27] is a class of vulnerable smart contracts without proper access control to the "`selfdestruct`" operations, which appear in around 20% of unique smart contracts [7]. Another high-profile victim due to vulnerable access control policy is the Parity Wallet. The wallet was hacked through two different attack vectors resulting in the stolen of $30M USD and the frozen of $100M USD, respectively. The Parity Wallet consumes a library contract, called WalletLibrary, to implement the basic functionality of a wallet. Anybody can deposit money into the wallet, but only the contract owner can withdraw the funds. The first attack vector exploited the unprotected initialization function to set the attacker as the contract owner and finally withdrew a large sum of money deposited by other wallet users. The root cause of this attack is that an unauthorized user can bypass the contract rule rather than the intended user access.

A difficulty in validating the conformance to such policies, i.e., whether the contract implementation adheres to the expected behaviors, is the lack of policy specifications. The current practice is to implement intended access control policies with ad-hoc

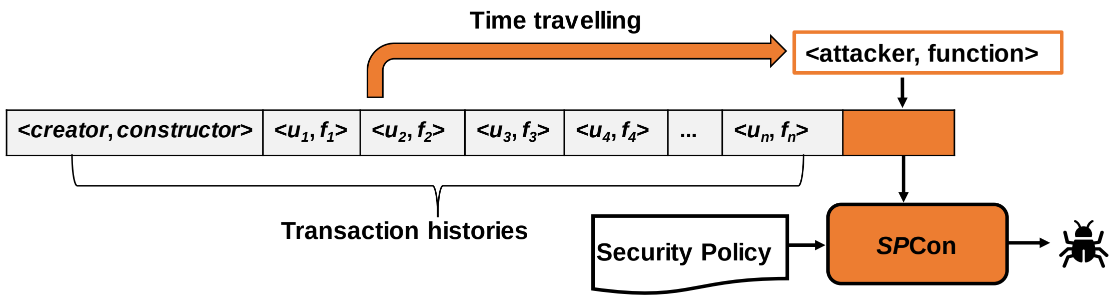


Figure 4.1: Illustration of the time-travel-based security policy validation.

Solidity [16] (i.e., the programming languages used to develop Ethereum smart contracts) idioms, such as the "`require`"statements, to check if the address of a user is in a predefined whitelist. Many security vulnerabilities mentioned earlier are results from this ad-hoc approach. In particular, when the number of roles and the complexity of access patterns increase, it becomes fairly easy for contract developers to make mistakes and introduce bugs causing security vulnerabilities.

In this chapter, we would like to address this problem by relying on past transactions of a contract to recover a *likely* access control model, which can then be validated against various security rules and identify potential bugs related to user permissions. We plan to implement a security policy validator powered by "time traveling" through transaction histories. As is shown in Fig. 4.1, the key idea is that, because of the transparency and immutability of blockchain transactions, the transaction histories of a smart contract application from its initial deployment up to today are always available on the blockchain. The historical transactions of a smart contract contain benign user behaviors, assuming the contract is not yet attacked by any malicious party. Due to the unique nature of DApps, we can safely assume that the history of a smart contract is *clean*, if it is still *alive*, i.e., not being destructed or abandoned. Therefore, we are able to reverse engineer an access control model which is permissible enough to subsume all historical transactions. Since historical transactions only under-approximate the behaviors allowed by the contract implementation, by validating the recovered model against the actual contract implementation, we will be able to discover discrepancies, indicating potential future violations of security policies.

```
1  pragma solidity ^0.5.0;
2
3  contract Dicether{
4
5    address owner, server;
6    bool activated = false,  paused = true;
7    enum Game{ENDED, ACTIVE, USECANCEL, SERVERCANCEL};
8    mapping(address=>Game) games;
9
10   constructor(address _serverAddress) public
11   {
12       owner = msg.sender;
13       server = _serverAddress;
14   }
15   // Pause contract
16   function pause() public onlyOwner onlyNotPaused
17   {
18       paused = true;
19   }
20   // Unpause contract
21   function unpause() public onlyOwner onlyPaused onlyActivated
22   {
23       paused = false;
24   }
25   function activate() public onlyOwner onlyNotActivated
26   {
27       activated = true;
28   }
29   function transferOwnership(address _newOwner) public onlyOwner
30   {
31       owner = _newOwner;
32   }
33   function transferProfitToHouse() public{
34     // transfer profit
35   }
36
37   // Create game
38   function createGame() public onlyNotPaused payable
39   {
40       require(games[msg.sender]==ENDED);
41       games[msg.sender] = ACTIVE;
42   }
43   // Cancel game
44   function userCancelGame() public payable
45   {
46       require(games[msg.sender]!=ENDED);
47       if (games[msg.sender]==ACTIVE){
48           games[msg.sender] = USECANCEL;
49       }else if (games[msg.sender]==SERVERCANCEL){
50           games[msg.sender] = ENDED;
51       }else{
52           revert();
53       }
54   }
55   // Cancel game
56   function serverCancelGame(address _userAddress) public
   onlyServer
57   {
58       require(games[_userAddress]!=ENDED);
59       if (games[_userAddress]==ACTIVE){
60         games[_userAddress] = SERVERCANCEL;
61       }else if (games[_userAddress]==USECANCEL){
62         games[_userAddress] = ENDED;
63       }else{  revert(); }
64   }
65   // End game
66   function serverEndGame(address userAddress) public onlyServer
67   {
68       require(games[_userAddress]==ACTIVE);
69       games[_userAddress] = ENDED;
70   }
71
72 }
```

FIGURE 4.2: The abstracted Solidity contract of Dicether.

4.2 Motivating Example

Dicether is an contract based casino decentralized application running on Ethereum. It uses a smart contract based state channel implementation to try to provide a fast, secure and fair gambling experience [74]. Figure 4.2 shows the abstracted version of *Dicether*. The constructor function (Line 10) sets the creator, namely *msg.sender* as the contract owner, which is responsible to pause/unpause contract in case of emergency and specifies the server *server* who administrates all games.

In most cases, game players invoke *createGame* to create a gambling game (Line 37) with ether deposition as the game stake, and once a game approach the deadline, the server will end the game via *serverEndGame* (Line 65) and also distribute the profit to the game player. Alternatively, the players can cancel their games via *userCancelGame* (Line 43). If the server also confirm the cancellation via *serverCancelGame*, the game will be ended. Once the previous game has been ended, the players can create new games and follow the aforementioned procedure as well.

The access control of *Dicether* is complicated. On the one hand, there are server types of users to perform different tasks such as the *owner*, *server*, *players*. On the other hand, tasks performed by users should follow a predefined temporal logic, e.g., a player must first create a game, then he/she can cancel the game. Besides, there exists a lot of uncertainty about the composition of users involved in a running Dicether contract. First, the exact *owner* and *server* is unknown before contract deployment. Second, the intended user behavior is unknown even there is specification about smart contracts and the intended user behavior is usually understood only after large-scale use.

The running *Dicether* contract has more than 9,427 transactions sent by around 400 different users (an Ethereum account can be a user) as of May 2021. Since *Dicether* is still active, we incline to believe the contract runs as expected by developers and users. Further, the set of user behaviors in its history provides sufficient knowledge to recover its access control model, i.e., user permission model where includes all intended user behavior.

4.3 User Permission Model

In this section, we will present user permission analysis framework. We base our user permission model on role-based access control (RBAC) which is a general framework for implementing various security policy. We focus on three types of information-flow security policy. (1) Integrity. The users of low security-level role cannot write to information managed by the users of high security-level role. (2)

Separation of duty. There is no information flow between two or more user roles. (3) Cardinality. There is a constraint on the range of roles who can participant in a session.

4.3.1 Role-Based Access Control Model

Role-based Access control (RBAC) has been well studied in the last twenty years since the establishment of the NIST RBAC standard in 1995 [75]. We borrow the standard RBAC definition [76] here.

Definition 4.1. RABC. The general RBAC model M can be defined in a tuple (U, R, P, S, RH, PA, UA). U, R, P and S are users, roles, permissions and sessions respectively. U, R and P are users, roles and permissions respectively. $RH \subseteq R \times R$ is a partial order on R called the role hierarchy, usually written as "$\succ$". $PA \subseteq P \times R$ is the permission-to-role assignment relation while $UA \subseteq U \times R$ the user-to-role assignment relation. And $U \times S \times 2^{R_{active} \subseteq \{r | x \in U \wedge (x,r) \in UA\}}$ is the role activation for each user in a session, where a session is the unit of access control.

Notice that RBAC is policy neutral and can implement different security policies mainly owing to the generality of permission concept. Similarly, the context of a session varies a lot between different contracts.

Definition 4.2. Session. A session is actually the unit of multi-party communication between users and contract. A session is usually implemented as a stateful resource of smart contract where each session is assigned a unique identifier, i.e., *id*. Meanwhile, a session transits to a state by user activities, i.e., user calling contract functions which take *id* as either input or output.

Definition 4.3. Role-Based Session Finite State Machine. A session usually has a limited number of states which can be interpreted by a finite state machine. A role-based session finite state machine, *SFSM*, can be defined as a 5-tuple (S, s_0, F, L, T). S is a set of finite states of sessions, $s_o \in S$ is the initial state, and $F \subseteq S$ is the set of final states. L is the set of user activities, $(role, action) \in L$ while T is the set of transitions and $T \subseteq S \times L \times S$. And *role* is defined in Definition 4.1 while *action* is the invocation of a concrete contract function.

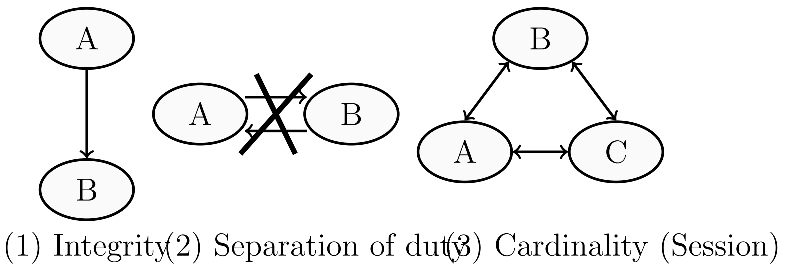


FIGURE 4.3: Overview of security policy.

4.3.2 Security Policies on Top of User Permissions

In this section, we present an overview of security policies: (1) integrity, (2) separation of duty, and (3) cardinality in Fig. 4.3. All security policies focus on information flow between roles. Note that each arrow represents information flow direction and ellipse represents user role. Information integrity requires information flows from the role of higher security level to the role of low security level. And separation of duty requires no information flows between two roles. Cardinality requires no other roles can involve in the information flow inside session.

For static enforcement of security policy, most information-flow properties are linear-time properties. However, LTL cannot express these properties because LTL can only express trace properties while information-flow properties need the comparison of multiple execution traces [77]. In [77] the authors proposed HyperLTL which extends LTL with trace quantifiers in order to relate multiple execution traces. The grammar of HyperLTL is listed below, where $a \in AP$, AP is a finite set of propositional variable; and π ranges over trace variables.

$$\begin{aligned}\varphi ::= \quad & true \;\mid\; a_\pi \;\mid\; \neg\varphi \;\mid\; \varphi \vee \varphi \;\mid\; \varphi \wedge \varphi \\ \mid\; & \bigcirc\varphi \;\mid\; \varphi U \varphi \;\mid\; \varphi R \varphi \;\mid\; \exists\pi.\varphi \;\mid\; \forall\pi.\varphi\end{aligned}$$

Moreover, the usual derived temporal operators *eventually* $\Diamond\varphi \equiv true\ U\ \varphi$ and *globally* $\Box\varphi \equiv \neg\Diamond\neg\varphi$. The following HyperLTL property specifies integrity for a high security-level role h: $\forall\pi.\forall\pi'.\Box(I_{h,\pi} = I_{h,\pi'}) \implies \Box(O_{h,\pi} = O_{h,\pi'})$, where I and O are the sets of propositions for public inputs and outputs. And $I_{h,\pi} = I_{h,\pi'}$ and $O_{h,\pi} = O_{h,\pi'}$ indicates that the atomic propositions I_h and O_h are equal in the path π and π' at the current point of time. This property implies that if the input I_h of high security-level user is same on any two traces, the output O_h of high security-level user remains unchanged.

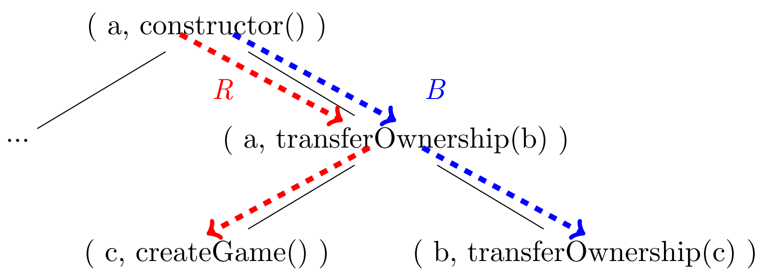


FIGURE 4.4: Execution tree of Dicether

However, this strong integrity property does not apply to the dynamic enforcement of security policy of smart contract since the user of high security-level role can announce its privilege and promote the user of low security-level role to be of high security-level role. For example, HyperLTL integrity property fails for Dicether. As shown in fig. 4.4, assuming that there are three users: a, b and c. Initially a is of high security level who calls the constructor function of Dicether which sets he/she as the contract owner. Take the value of contract owner as the output of a, the proposition on the output is a constraint on the owner value. the red path shows an execution trace R: "a calling constructor function. $\rightarrow$ a transfers ownership to b. $\rightarrow$ c create a game". At the end of trace R, the contract owner is b. And the blue path shows an execution trace B: "a calling constructor function. $\rightarrow$ a transfers ownership to b. $\rightarrow$ b transfers ownership to c". At the end of trace B, the contract owner is c. The two execution traces R and B have same input of a but the output, i.e., contract owner of a is b of R and c of B respectively. Apparently, the HyperLTL integrity property is violated but *transferownership* has no security risk for that. The root cause is that the HyperLTL property is not suitable for describing dynamic security policy of smart contracts.

Integrity. Therefore, we weakened the HyperLTL property of integrity into a standard LTL property in this work. The following LTL propery φ_1 specifies weakened integrity for a high security-level role h for smart contract having history Γ:

$$\forall \pi^+.\Box(I_{h,\Gamma} = I_{h,\Gamma \triangleleft \pi^+}) \implies \Box(O_{h,\Gamma} = O_{h,\Gamma \triangleleft \pi^+})$$

The LTL property can be simplified as $\Box(I_h^0 = I_h) \implies \Box(O_h^0 = O_h)$, where I_h^0 and O_h^0 respectively represent the input and the final output of high security-level role on Γ. Suppose we have a contract model M at the time of Γ, the problem whether $M \implies \varphi_1$ is of our interest.

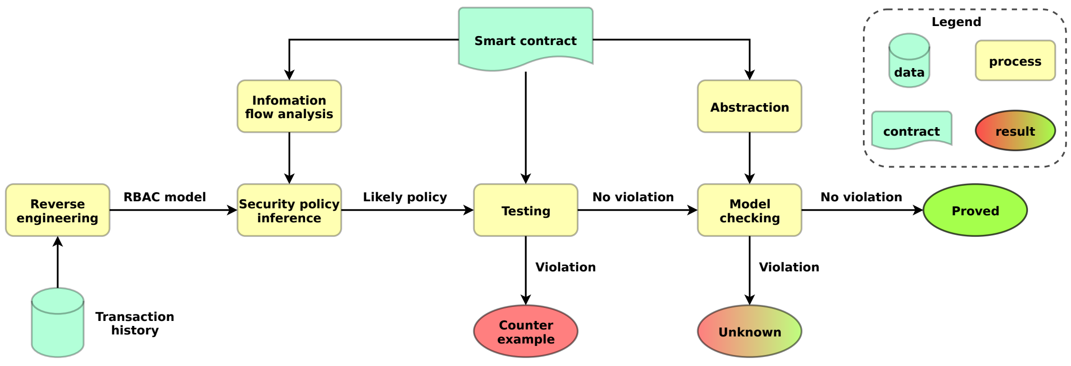


FIGURE 4.5: Workflow of the SpCon Framework.

Separation of duty. Similarly, the following LTL property φ_2 specifies separation of duty given π_{R_s} ranging over execution traces by a set of roles R_s:

$$\bigwedge_{r \in R_s} \Box(I_r^0 = I_r) \implies \Box(O_r^0 = O_r)$$

Cardinality (Session). The following LTL property φ_3 specifies cardinality for a set of roles within session R_c:

$$\Box(\bigwedge_{r \in R_c} I_r^0 = I_r) \implies \Box(\bigwedge_{r \in R_c} O_r^0 = O_r)$$

4.4 Framework

In this chapter, we present SpCon, a framework for finding user permission bugs via time travel on transaction history of smart contracts. SpCon consists of four main stages: (1) RBAC model construction from transaction history, (2) Security policy inference via information flow analysis of smart contracts, (3) Testing and (4) Model checking for finding permission bugs and proving security policy respectively.

4.4.1 The User Role Mining

To construct the user model of a smart contract, we need first to identify user roles. We do this by reverse engineering user roles from the history of the smart contract. Firstly, we extract user access information where users have successfully called contract functions. For simplicity, a user access is a 2-tuple (u, f) where $u \in U$ and $f \in ABI$, where ABI is the application binary interface containing public functions of smart contracts. In this paper, we use ABI to only denote public functions which can be called by users and change contract state at the same time. U consists of different external Ethereum accounts calling any of the ABI functions of the smart contract. All the user-function pairs represent the overall user accesses in the history. The extraction of the list of user-function pairs, UF, is illustrated using the rule in Fig. 4.6. With user-function pairs UF, we can mine user roles.

$$\frac{(u_1, val_1, \mathrm{f}_1, 1), \ldots, (u_n, val_n, \mathrm{f}_n, 1)}{\begin{array}{c} U \leftarrow \{u_1, \ldots, u_n\} \quad P \leftarrow \{\mathrm{f}_1, \ldots, \mathrm{f}_n\} \\ UF \leftarrow \{(u_1, \mathrm{f}_1), \ldots, (u_n, \mathrm{f}_n)\} \end{array}} \text{[Extraction]}$$

FIGURE 4.6: The extraction of user accesses.

Role mining aims to utilize existing user access to reason about user roles R. This role mining problem (RMP) has been well-defined by Vaidya and et al. [78]. For RMP, each mined role consists of users and is assigned to some functions. The goal of RMP is to group the users based on their access function sets, namely that the users having the same function sets will be grouped as a role. We use a lattice-based role approach [79] to mine the basic user roles from the function pairs UF. Each role is associated with a function set and consists of the users who have called the contract functions. However, the history of a smart contract is incomplete. Although the aforementioned approach achieves consistency with observed user accesses, the mined user roles could be an under-approximation of their actual roles, which encompass the sets of *potentially* accessible functions per user.

To obtain an over-approximation, we merge the mined basic user roles using the frequency of user accesses. To do that, we need first to establish role similarity. We adopt the widely used cosine similarity measure, which is calculated by:

$$f_i = \vec{ABI}[i] \tag{4.1}$$

$$\vec{V_1} : V_1[i] = \frac{\sum_{u_j \in R_1} n_{(u_j, f_i) \in UF}}{||R_1||} \tag{4.2}$$

$$\vec{V_2} : V_2[i] = \frac{\sum_{u_j \in R_2} n_{(u_j, f_i) \in UF}}{||R_2||} \tag{4.3}$$

$$\text{similarity}(R_1, R_2) = \cos(\vec{V_1}, \vec{V_2}) \tag{4.4}$$

where $n_{(u_j, f_i) \in UF}$ refers to the number of times the observation of user-function pair (u_j, f_i) occurred in the history represented by UF. We merge two roles with high

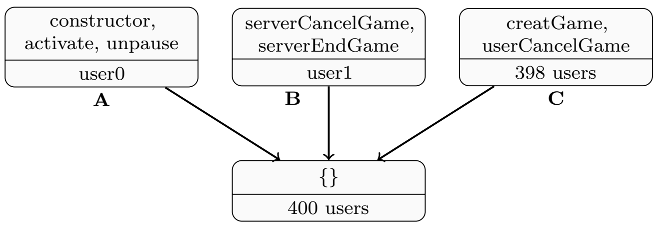

constructor, activate, unpause
user0
A
serverCancelGame, serverEndGame
user1
B
creatGame, userCancelGame
398 users
C
{}
400 users

FIGURE 4.7: The roles of Dicether.

similarity and thus achieve an over-approximation of users in the history. Here, we present the definition of the μ-similarity role mining problem.

Definition 4.4. μ-Similarity Role Mining Problem (μ-RMP). Given a set of users U, a set of functions P and a set of user function pairs UF_{set}, find a set of roles R, a user-to-role assignment matrix UA and a role-to-function assignment matrix PA by minimizing the number of roles $||R||$ such that

$$UF_{set} \subset UA \times PA$$
$$\text{similarity}(R_i, R_j) \leq \mu, \ \forall R_i, R_j \in R$$

where "$\times$" refers to matrix multiplication whose result is interpreted as a set of user-function pairs.

Figure 4.7 shows the mined user roles of Dicether. Each node represents a role consisting of users at the bottom and associated with different functions. It is clear that there are three type of separate roles, A, B and C. Further, the roles interaction relationship will be investigated in the next section.

4.4.2 SFSM Learning

Role-based session finite state machine is inferred via passive learning on the role behaviors within sessions. To do this, we need to identify sessions and user behaviors within sessions from transaction history. In smart contracts, session is usually explicitly declared in each function input (e.g., token id for NFT contracts) or implicitly identified by user address. And we can (semi-)automatically identify sessions with these prior knowledge. The user behaviors within each session will be further abstracted into role behaviors within each session which can avoid the infinity of user identity. Each session has only a role behavior sequence which

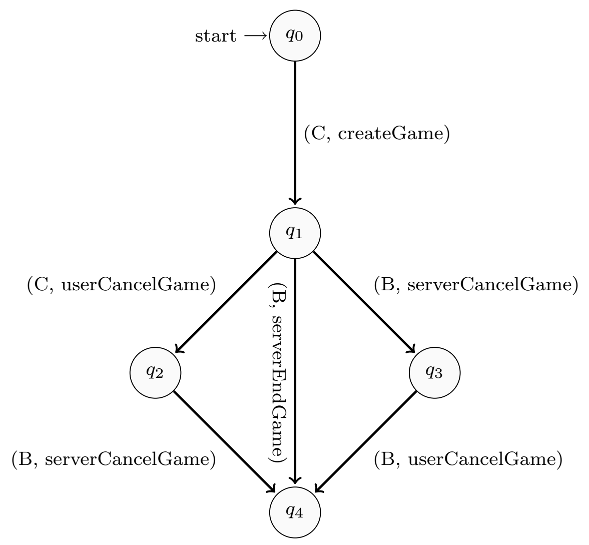


FIGURE 4.8: The SFSM of Dicether.

consists of ordered role behaviors in the term of their appearance. Finally, lots of role behavior sequences are collected from different sessions. Note that these sequences are all from successful transactions. Therefore, these sequences construct as positive examples for passive automata learning. It is difficult to collect negative examples because to prove a function is unreachable for users is infeasible in limited transaction history.

Figure 4.8 shows the role-based session finite state machine of Dicether. There are five states in total and two roles, C and B cooperate in gambling games. Moreover, the likely temporal property of SFSM can also be identified, such as CTL property: $\mathbf{AX}(createGame \implies serverEndGame \vee userCancelGame \vee serverCancelGame)$, though it falls outside of the scope of information security policy studied in this paper.

4.4.3 Security Policy Inference

Roles are associated with functions. We perform control-flow dependency analysis between functions, which can demonstrate the control-related information flow between roles. Briefly speaking, for each function, our interests are the state variables used on path conditions and the state variables changed at the end. All the control-flow dependency between functions determine the information flow between roles. We say information flows from role *A'* to *B'* if and only if a path of a *A''*s function is dependent on the state variables changed by one of a *B''*s

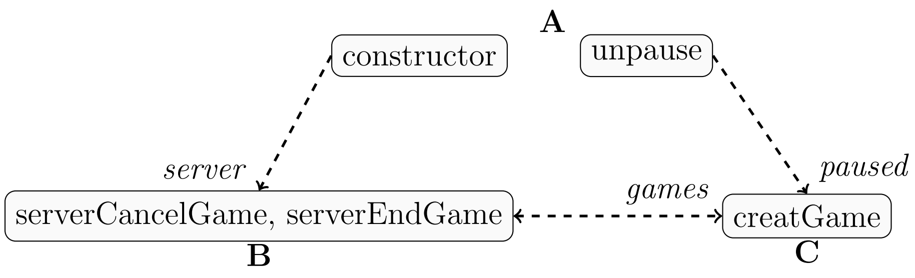


FIGURE 4.9: The information flow between roles of Dicether.

function. If information can only flow from *A'* to *B'*, information integrity should be satisfied between these two roles. If no information flows between *A'* and *B'*, separation of duty is satisfied between the two roles. Further, if information flow within sessions is only allowed over a fixed number of roles, cardinality is implied.

Figure 4.9 shows information flow between the three aforementioned roles of Dicether. Role B depends on the *server* specified by role A in constructor function, while role C depends on the value of *paused* as well. And only role B and C communicate within game sessions. Hence, there exists several likely security policy, i.e., (1) Integrity, $\Box(role(msgsender)! = A) \implies \Box(server_0 = server)$ and $\Box(role(msgsender)! = A) \implies \Box(paused_0 = paused)$ and (2) Cardinality, $\Box(role(msgsender)! = B \wedge role(msgsender)! = C) \implies \Box(games_0 = games)$.

4.4.4 Testing

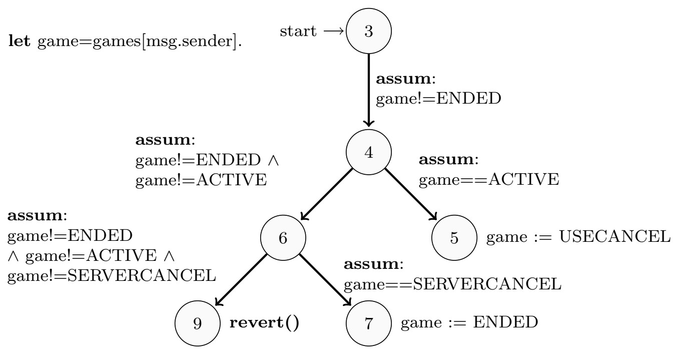
The goal of testing is to detect vulnerabilities on the aforementioned security policies.

Test Oracle. Given a integrity property φ: $\Box(I_h^0 = I_h) \implies \Box(O_h^0 = O_h)$, the testing attempts to generte an counterexample which satisfies $!\varphi$: $\Box(I_h^0 = I_h) \wedge \Diamond(O_h^0! = O_h)$. However, it is difficult to automatically test this temporal property because testing usually relies on atomic propositions as the test oracles, such as reentrancy [80], delegate call [6] and etc.. To do that, we need to reduce φ into practical test oracle t. The $\Box(I_h^0 = I_h)$ can be eliminated by $role(msgsender)! = h$ without loss of precision, where *msgsender* is the caller to contract functions. Then, $\Diamond(O_h^0! = O_h)$ could be over-approximated by function reachability propositions $\overset{f_o}{\vee} f_o$, where f_o refers to the function that could change O_h. For separation of duty and cardinality, the approach is similar.

```
while( true ){
    contract = replay( history );
    for( i: 0..k ){
        user, function = tcgeneration( );
        reach = symexec( contract, user, function );
        checkAndReport( reach );
}}
```

FIGURE 4.10: Smart contract testing

```
1   function userCancelGame() public payable
2   {
3     require(games[msg.sender]!=ENDED);
4     if (games[msg.sender]==ACTIVE){
5       games[msg.sender] = USECANCEL;
6     }else if (games[msg.sender]==SERVERCANCEL
     ){
7       games[msg.sender] = ENDED;
8     }else{
9       revert();
10    }
11  }
12
```



let game=games[msg.sender].
start
3
assum: game!=ENDED
4
assum: game!=ENDED ∧ game!=ACTIVE
assum: game==ACTIVE
6
5
game := USECANCEL
assum: game!=ENDED ∧ game!=ACTIVE ∧ game!=SERVERCANCEL
assum: game==SERVERCANCEL
9
revert()
7
game := ENDED

FIGURE 4.11: The control flow graph of Dicether's *userCancelGame*.

Test Environment. As for shown in fig. 4.10, first SPCON will replay all transaction history of smart contract to keep contract state close to its running environment on Ethereum.

Test Case Generation Strategy. Because it is infeasible to generate every possible transaction sequences, we limit the capability of SPCON to generate transaction sequence of length up to k. Further, the functions in a transaction sequence can also be partially ordered according to their control-flow dependency relationship, which can reduce unnecessary test cases. Then, SPCON uses symbolic execution to under-approximate the function reachability problem. Finally, SPCON checks and reports an counterexample if there is violation against the aforementioned test oracles.

4.4.5 Abstraction and Model Checking

Definition 4.5. Contract Model. A contract model M can be defined by (δ_0, F, Δ, T) where $\delta_0 \in \Delta$ is initial state of contract, $f \in F$ is a public function of contract and $T \subseteq \Delta \times F \times \Delta$ is transition function of contract state.

The initial state of contract is actually the abstraction of contract state at current transaction history. As stated before, only successful function invocation can change contract state. Each path of function can be abstracted by a boolean program: $Path(f) \implies Precondition() \wedge StateChanges(\Delta_1)$, where *Precondition()* is a set of satisfied predicates and *StateChanges(Δ_1)* is a set of assignments of state variables. Figure 4.11 shows the control flow graph of Dicether's *userCancelGame* function. The program path "Line 3 $\rightarrow$ Line 4 $\rightarrow$ Line 5" can be abstracted by "***assume**: game==Active; game:=USERCANCEL*". That means when the game is at active status, user can call *userCancelGame* to transit the game to user cancellation status.

It is impractical to get a precise abstraction of all contract semantic. For example, the inter-contract call inside functions is hard to abstract because there exists state explosion problem for a cross-contract model even all the inter-contract calls are statically determined. Therefore, in this paper, we derive a safe abstraction following interpolant lemma [81] to get an over-approximated abstract model for smart contracts. We say it is a safe abstraction if the property is true on the original model then it is true on the abstract model. We argue the over-approximated abstract model is also sound since the information security property relates to the state changes of smart contracts, which is precisely modeled instead.

Finally, the abstract model and security properties are verified by the state-of-art symbolic model checking tool, e.g., NuSMV [82]. If property holds, we say this property is proved and can further be added to the specification of smart contracts. Otherwise, we can symbolically execute or manually check the generated counterexample by model checking to confirm whether it is a true counterexample.

Chapter 5

Conclusion and Future Work

This report presents an thesis overview on security and reliability of smart contracts. The report mainly compiles the published works that I have done on fairness verification (see Chapter 2), model based testing and fuzzing (see Chapter 3) of smart contracts. The promising results of these works highlight the importance of fairness problem related to smart contracts, which usually deviates from the expectation of novice contract users respectively, and demonstrate the efficiency of model-based testing in coverage and implementation error detection, and witness the power of semantic test oracle. The current access control analysis work aims to find user permission bugs via time travel on transaction history of smart contracts. Its analysis framework is also illustrated in Chapter 4 and the evaluation will be finished in the near future.

5.1 Future Work

Meanwhile, there are other unaddressed research topics outlined in fig. 1.2 of Chapter 1.

- **Runtime Verification**. Runtime verification is a lightweight technique sitting between formal verification and testing. There are many runtime verification tools [83, 84] for the detection of predefined vulnerable patterns via the code instrument to EVM clients [83] or smart contracts [84]. However, these predefined vulnerable patterns are often simple and usually not precise and the code instrument will incur other cost such as the blockchain fork and increased gas consumption. On the other hand, they are not suitable for the verification of dynamic security policy. Security policy of smart contracts can be changed at the runtime, which is hard to be covered by predefined patterns. Therefore, the future work, runtime verification will target dynamic security policy and this will be implemented via oracle mechanisms to ensure smart contract security on-the-fly.

- **Specification Mining**. Although a considerable number of formal specification techniques have been proposed in contract level or program level [22], there are very limited smart contracts that is well specified or documented [63, 71]. Without specification, the evaluation of smart contract is relatively restricted for common security concern or fairness concern, which needs domain knowledge instead. In this future work, specification will be mined from smart contracts, either from its implementation or its running logs. We hope specification mining could facilitate the better evaluation of smart contracts in the research community.

Bibliography

[1] David Siegel. *Understanding the DAO Attack*, 2016. URL `https://www.coindesk.com/understanding-dao-hack-journalists`. 2

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

[3] Etherscan. `https://etherscan.io`, 2020. 3, 8

[4] Ethereum transaction fees fall by 75% as congestion eases. `https://cointelegraph.com/news/ethereum-transaction-fees-fall-by-75-as-congestion-eases`, 2020. 3, 35

[5] FISCO BCOS. `https://fisco-bcos.org/`, 2020. 3, 35, 36

[6] 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. 3, 31, 44, 58

[7] Oyente. `https://github.com/melonproject/oyente`, 2019. An Analysis Tool for Smart Contracts. 31, 44, 47

[8] *Securify*. Sofware Reliability Lab, 2019. URL `https://securify.ch/`. 31

[9] *SmartCheck*. SmartDec, 2019. URL `https://tool.smartdec.net`.

[10] 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. 3, 31, 41, 44

[11] 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`. 4

[12] 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. 4, 33

[13] Matthew O Jackson. Mechanism theory. *Available at SSRN 2542983*, 2014. 7, 12, 33

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

[15] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. 8, 11

[16] Solidity. Solidity. `https://solidity.readthedocs.io/en/v0.5.1/`, 2018. 9, 37, 48

[17] 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

[18] 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, 21, 22

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

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

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

[22] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification, 2020. 31, 62

[23] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 243–269. Springer, 2018. 31

[24] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? feb 2019. URL `http://arxiv.org/abs/1902.06710`. 31

[25] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proc. ACM Program. Lang.*, 3 (OOPSLA), October 2019. doi: 10.1145/3360615. URL `https://doi.org/10.1145/3360615`. 31

[26] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM, 2016. 31, 44

[27] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663. ACM, 2018. 31, 47

[28] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 825–841. IEEE Computer Society, may 2020. 31

[29] Yu Feng, Emina Torlak, and Rastislav Bodik. Precise Attack Synthesis for Smart Contracts. *arXiv preprint arXiv:1902.06067*, 2019. 31

[30] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, pages 442–446, Feb 2017. doi: 10.1109/SANER.2017.7884650. 31

[31] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, page 116, 2018. 31

[32] Joshua Ellul and Gordon J Pace. Runtime verification of ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 158–163. IEEE, 2018. 31

[33] Mythril. `https://github.com/ConsenSys/mythril`, 2019. A Security Analysis Tool for EVM Bytecode. 31, 44

[34] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing safety of smart contracts. In *The Network and Distributed System Security Symposium*, 2018. 31, 44

[35] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018. 44

[36] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. scompile: Critical path identification and analysis for smart contracts. *arXiv preprint arXiv:1808.00624*, 2018. 44

[37] Haijun Wang, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. Oracle-supported dynamic exploit generation for smart contracts, 2019. 31

[38] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016. 31

[39] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. Kevm: A complete semantics of the ethereum virtual machine. Technical report, 2017. 31

[40] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 912–915. ACM, 2018. 31

[41] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE symposium on security and privacy (SP)*, pages 1695–1712. IEEE, 2020. 32

[42] Jiao Jiao, Shang-Wei Lin, and Jun Sun. A generalized formal semantic framework for smart contracts. In *2020 International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 75–96, 2020. 32

[43] Tesnim Abdellatif and Kei-Léo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2018. 32

[44] Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. 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)*, pages 980–987. IEEE, 2018. 32

[45] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. Learning fair representations. In *International Conference on Machine Learning*, pages 325–333, 2013. 32

[46] Anupam Datta, Shayak Sen, and Yair Zick. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *2016 IEEE symposium on security and privacy (SP)*, pages 598–617. IEEE, 2016.

[47] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. Fairsquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages*, 1:80:1–80:30, 2017. 32

[48] Sahil Verma and Julia Rubin. Fairness definitions explained. In *IEEE/ACM International Workshop on Software Fairness (FairWare)*, pages 1–7, 05 2018. doi: 10.1145/3194770.3194776. 32

[49] Aws Albarghouthi and Samuel Vinitsky. Fairness-aware programming. In *the Conference on Fairness, Accountability, and Transparency*, pages 211–219, 2019. 32

[50] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *Proceedings of the 2018 World Wide Web Conference*, pages 1409–1418. International World Wide Web Conferences Steering Committee, 2018. 33

[51] Eric S Maskin. Mechanism design: How to implement social goals. *American Economic Review*, 98(3):567–76, 2008. 33

[52] Paul Klemperer. Auction theory: A guide to the literature. *Journal of economic surveys*, 13(3):227–286, 1999. 33

[53] Daniel Lehmann, Liadan Ita Oćallaghan, and Yoav Shoham. Truth revelation in approximately efficient combinatorial auctions. *Journal of the ACM (JACM)*, 49(5):577–602, 2002. 33

[54] Adam Hahn, Rajveer Singh, Chen-Ching Liu, and Sijie Chen. Smart contract-based campus demonstration of decentralized transactive energy auctions. In *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pages 1–5. IEEE, 2017. 33

[55] Yi-Hui Chen, Shih-Hsin Chen, and Iuon-Chang Lin. Blockchain based smart contract for bidding system. In *2018 IEEE International Conference on Applied System Invention (ICASI)*, pages 208–211. IEEE, 2018. 33

[56] Hisham S Galal and Amr M Youssef. Succinctly verifiable sealed-bid auction smart contract. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 3–19. Springer, 2018. 33

[57] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. A smart contract for boardroom voting with maximum voter privacy. In *International Conference on Financial Cryptography and Data Security*, pages 357–375. Springer, 2017. 33

[58] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, pages 142–161. Springer, 2015. 33

[59] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. Quantitative analysis of smart contracts. In *European Symposium on Programming*, pages 739–767. Springer, Cham, 2018. 33

[60] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008. 35

[61] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014. 35

[62] Open, proven, enterprise-grade dlt. `https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf`, 2020. 35

[63] Azure blockchain workbench. `https://azure.microsoft.com/en-us/features/blockchain-workbench/`, 2020. 35, 62

[64] Webank (china). `https://en.wikipedia.org/wiki/WeBank_(China)`, 2020. 36

[65] The progressive javascript framework. `https://vuejs.org/`, 2020. 38

[66] Node.js. `https://nodejs.org/`, 2020. 38

[67] Haijun Wang, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. Oracle-supported dynamic exploit generation for smart contracts. *arXiv preprint arXiv:1909.06605*, 2019. 41, 44

[68] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018. 42, 44

[69] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68. ACM, 2018. 44

[70] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. *arXiv preprint arXiv:2004.08563*, 2020. 44

[71] Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers*, volume 12031, page 87. Springer Nature, 2020. 44, 62

[72] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. jan 2019. URL `http://arxiv.org/abs/1901.01292`. 44

[73] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012. 44

[74] Dicether. Dicether: A secure dice game, 2018. URL `https://dicether.github.io/paper/paper.pdf`. 49

[75] Ravi Sandhu, David Ferraiolo, Richard Kuhn, et al. The nist model for role-based access control: towards a unified standard. In *ACM workshop on Role-based access control*, volume 10, 2000. 51

[76] Ravi S Sandhu. Role-based access control. In *Advances in computers*, volume 46, pages 237–286. Elsevier, 1998. 51

[77] Markus N Rabe. A temporal logic approach to information-flow control. 2016. 52

[78] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 175–184, 2007. 55

[79] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin Calo, and Jorge Lobo. Mining roles with semantic meanings. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 21–30, 2008. 55

[80] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016. 58

[81] William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957. 60

[82] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *International conference on computer aided verification*, pages 495–499. Springer, 1999. 60

[83] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. Soda: A generic online detection framework for smart contracts. In *27th Ann. Network and Distributed Systems Security Symp.* The Internet Society, 2020. 61

[84] Sefa Akca, Ajitha Rajan, and Chao Peng. Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489. IEEE, 2019. 61