# Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications

Yue Duan*
Illinois Institute of Technology &
University of Utah
Chicago, IL, USA
yduan12@iit.edu

Xin Zhao
Nanjing University
Nanjing, China
zhao_xin@smail.nju.edu.cn

Yu Pan
University of Utah
Salt Lake City, UT, USA
yupan97@cs.utah.edu

Shucheng Li
Nanjing University
Nanjing, China
shuchengli@smail.nju.edu.cn

Minghao Li
Harvard University
Boston, MA, USA
minghaoli@g.harvard.edu

Fengyuan Xu
National Key Lab for Novel Software
Technology, Nanjing University
Nanjing, China
fengyuan.xu@nju.edu.cn

Mu Zhang
University of Utah
Salt Lake City, UT, USA
muzhang@cs.utah.edu

## ABSTRACT

We propose VetSC, a novel UI-driven, program analysis guided model checking technique that can automatically extract contract semantics in DApps so as to enable targeted safety vetting. To facilitate model checking, we extract *business model graphs* from contract code that capture its intrinsic business and safety logic. To automatically determine what safety specifications to check, we retrieve textual semantics from DApp user interfaces. To exclude untrusted UI text, we also validate the UI-logic consistency and detect any discrepancies. We have implemented VetSC and applied it to 34 real-world DApps. Experiments have demonstrated that VetSC can accurately interpret smart contract code, enable autonomous safety vetting, and discover safety risks in real-world Dapps. Using our tool, we have successfully discovered 19 new safety risks in the wild, such as expired lottery tickets and double voting.

## CCS CONCEPTS

• **Security and privacy → Software and application security**.

## KEYWORDS

decentralized apps; smart contracts; safety verification; semantics

**ACM Reference Format:**
Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. 2022. Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3559384

*This work was conducted when the first author was a postdoc at the University of Utah under the supervision of Mu Zhang. Yue Duan, Fengyuan Xu and Mu Zhang are corresponding authors.



*CCS '22, November 7–11, 2022, Los Angeles, CA, USA*

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3559384

## 1 INTRODUCTION

A decentralized application (DApp) is an application running on a decentralized network. It combines a smart contract – a program atop blockchains – and a front-end user interface. Compared to raw smart contracts, which require professional skills to interact with [21], DApps hit a sweet spot between trustworthy transactions and user-friendly interfaces. Hence, DApps have gained increasing popularity – 4,000 DApps of 18 categories have been created in the past three years, and hundreds of new apps are being released monthly [42]. Zethyr Exchange [62], one of the most active DApps, can have 77K daily users and a total of 3.4 million unique users.

Nonetheless, the embrace of decentralized apps has also brought safety risks to the rapidly growing audience. Unsafe DApps may cause severe financial losses [17] due to incorrect [45] or unfair [33] smart contract logic. For instance, the integrity of a "no-reserve" auction, where the item for sale must be sold regardless of price, will be ruined if it allows the seller to bid. Many previous studies [23, 26, 33, 34, 38, 39, 45, 50, 53, 56, 58] have sought to verify the correctness and safety of smart contracts to address these problems.

Unfortunately, the prior work requires tedious manual efforts and heuristics to understand the semantics of contracts and further create contract-specific safety specifications. In particular, to apply high-level safety specs (e.g., *"sellers must not bid"*) to corresponding low-level smart contract code, a safety verifier must *(a)* uncover the business logic-level actions of individual contract functions, such as *placing a bid*, and *(b)* associate transactional concepts (e.g., *bidder, seller, bid price*) to specific variables in functions, and then manually create contract-specific formal specs from the high-level specs. Prior work depends solely on developer-created function symbols and variable names in the source code to obtain these crucial semantics. However, these symbols are untrustworthy and

sometimes unavailable due to the lack of source code or intentional obfuscation. In addition, repetitive manual efforts to interpret contract source code must be made for every single contract function. Consequently, prior work cannot automatically apply high-level safety specifications to even the same type of contracts, as their implementation will likely use different functions or variable names.

To address this limitation, we automatically bridge the semantic gap to facilitate the DApp vetting. Automated and robust semantics recovery has been well studied in many areas, including virtual machine introspection [18, 25, 31], memory forensics [13, 16, 19, 36, 37], binary code analysis [15, 35] and mobile and IoT app analysis [44, 48, 57]. Yet none can solve the unique problem in the smart contract domain. VMI and binary analysis techniques can only recover OS-level data structures and primitive types, and do not speak to application-level semantics; mobile and IoT app analyzers interpret program behaviors based upon semantics-rich framework APIs (e.g., `sendTextMessage()`, `lock.unlock()`) which, however, are absent from smart contract programming languages.

In contrast, applying model checking to recovering high-level semantics is promising. Prior work [40] has utilized model checking to automatically match semantic-level industrial process specifications with low-level controller code. Nevertheless, this technique cannot be adopted in our problem for two reasons: *(1)* it has relied on an assumption that analysts have high-level yet precise knowledge of target industrial processes, and therefore can craft the exact specifications. However, this assumption does not hold when analyzing diverse DApps developed by unknown parties. *(2)* Compared to controller logic that strictly expresses core dependencies among limited control variables, smart contract code is less rigid and often includes variables and their relations that are irrelevant to its core logic. Introducing a large amount of non-essential code may cause excessive complexity and imprecision for model checking.

To solve these problems, we propose a novel *UI-driven, program analysis guided* model checking technique that can automatically extract contract semantics in decentralized apps to enable targeted safety vetting. To exclude irrelevant code and reduce search space for model checkers, we perform static program analysis to extract *business model graphs* from contract functions that can capture intrinsic business logic across critical transaction properties. To automatically select specifications for safety vetting, we leverage textual information collected from DApp user interfaces, describing high-level underlying contract logic behaviors.

Note that, however, while our observation is that UI information from benign developers is largely faithful and thus can assist us in understanding contract semantics, we do not assume that all DApp UI texts are correct or complete. In a similar vein, prior work [44, 48] has discovered an inconsistency between internal program logic and external textual descriptions. Hence, to use caution when incorporating UI information, we take a *proof-of-contradiction*-based approach. Particularly, we first consider the business logic explained by UI texts to be correct, which allows us to avoid an exhaustive search for corresponding specifications and thus will enable us to check a smart contract function against only the described logic. Then, if the internal logic, in fact, deviates from the DApp descriptions (e.g., UI shows a DApp is a gambling app but its internal smart contract code implements an auction logic), an alert will arise. Only if the contract logic matches the UI semantics, we then further check whether the contract complies with the *de facto* safety rules for the described business logic, such as the policies for participating in a bidding process [33] or canceling an auction [20].

We have developed a system, VetSC, to achieve our goal. Given a DApp, we *(A)* first extract the business model of every smart contract function. In particular, we identify key transaction-level properties in each function. Starting from these properties, we perform static control-flow and dataflow analyses to build *business model graphs* that represent the function's semantic model. *(B)* Next, we recover the semantics of distilled graphs via UI-guided model checking. More concretely, we analyze the DApp's web code to correlate each contract function to a front-end widget that triggers this function. We then apply natural language processing techniques to the texts on the widget, to infer the business logic the widget describes. Guided by the textual semantics extracted from the UI component, we can thus select the corresponding business logic specification from our predefined business models. We hence conduct model checking that verifies the *business model graphs* against this specification. We will report this issue and terminate our analysis if a major inconsistency is detected. *(C)* If the extracted graphs are consistent with the described semantics, we can then determine the business logic of the contract function. Thus, we can automatically apply corresponding safety rules to the vetting of this function to detect any violations. Note that manually crafting business logic and safety specifications is a one-time effort for each semantic category .

We have implemented VetSC in 3500 lines of Python code and applied it to 34 real-world DApps, containing 494 Solidity functions. To the best of our knowledge, this is the largest experimental dataset for detecting logic errors in dapps. Experimental results have demonstrated that VetSC outperforms the state-of-the-art technique and can effectively discover the safety risks in real-world DApps, where source code is either absent or heavily obfuscated.

In summary, this paper makes the following contributions:

- We develop a new technique that uniquely combines program analysis, model checking and UI interpretation to solve a novel and important problem – automated semantic-level safety analysis – in the new context of smart contracts.
- Our static analysis selectively extracts Solidity bytecode instructions that represent program semantics, as it is hard (if not impossible) for humans to precisely capture such core logic from many irrelevant instructions, obfuscated or closed-source programs.
- Our model checker maps high-level concepts to concrete variables, and thus automatically concretizes abstract safety policies for individual contracts using their variable names. On the contrary, prior work must manually create concrete specs for every contract.
- We identify auxiliary semantic information from DApp UI, which reduces the efforts to find possibly matching specifications.
- VetSC has discovered 19 new safety bugs in real-world apps, which cannot be automatically detected by the state-of-the-art techniques.

To facilitate further research, we have made the source code publicly available[1].

[1] https://github.com/vetsc/VetSC

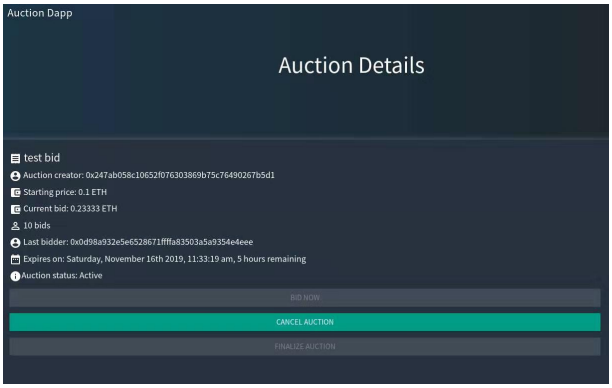


**Figure 1: Bid and Cancel Auction**

## 2 PROBLEM & APPROACH

We demonstrate the problem using a real-world Ethereum decentralized app, *Auction DApp*. Figure 1 illustrates its user interface with three buttons. Specifically, a user can click the *"BID NOW"* button to participate in an existing auction created by a merchandise owner. An auction may also be cancelled by its owner, if she presses the specific button *"CANCEL AUCTION"* on the same page. These two buttons are internally associated with two smart contract functions `bidOnAction()` and `cancelAuction()`, respectively.

### 2.1 Motivating Example

***Smart Contract Functions.*** Figure 2 depicts the two smart contract functions implemented using Solidity [54], the dedicated programming language for Ethereum contracts. Notice that we show the program in source code for the sake of readability while our analysis is performed directly on bytecode.

`bidOnAction()` exercises the bidding process. It acquires the bid price from the transaction property `msg.value` (Ln.2), which represents the amount sent by the function caller (i.e., bidder). Then, it performs two checks to ensure the owner cannot bid and the auction has not expired. If both checks are passed, the function compares the bid price with the current highest bid `lastBid.amount` or the default price `startPrice` depending on the existence of any prior bid, and thus determines if the new price is higher (Ln.11-21). If so, it will accept the new highest bidder by pushing it as a `Bid` object into the list (Ln.24-27).

`cancelAuction()` uses a *modifier* `isOwner` (Ln.31) to ensure that only the merchandise owner can call the function. When executed, the function first retrieves the `Auction` object `myAuction` and the array of previously accepted bids `accepted[_id]` based upon the given `_id` (Ln.32-33). Next, it refunds the last bidder in this array, which is essentially the current highest bidder, provided that there exist prior bids (Ln.36-39). In the end, this auction becomes inactive and a log will be generated via `emit` statement (Ln.40-41).

***Safety Risk.*** A safety problem in `cancelAuction()` can be triggered when the button *"CANCEL AUCTION"* is clicked. Two safety rules must be enforced when canceling an auction [20]: *(1)* only the owner can cancel her auction; and *(2)* no valid bid has been received. The second one guarantees the fairness of this auction as it ensures that an auction cannot be arbitrarily terminated when participants have already placed valid bids, regardless of the owner's intention

```
1  function bidOnAuction(uint _id) public payable {
2      uint256 ethAmountSent = msg.value;
3
4      // owner cannot bid on his/her own merchandise
5      Auction memory myAuction = auctions[_id];
6      if(myAuction.owner == msg.sender) revert();
7
8      // check whether auction has expired
9      if(block.timestamp > myAuction.deadline) revert();
10
11     // check whether previous bids exist
12     uint bidsLength = accepted[_id].length;
13     uint256 tempAmount = myAuction.startPrice;
14     Bid memory lastBid;
15     if(bidsLength > 0) {
16         lastBid = accepted[_id][bidsLength-1];
17         tempAmount = lastBid.amount;
18     }
19
20     // check if bid price is greater than the current highest
21     if(ethAmountSent < tempAmount) revert();
22
23     // add the new bid to auction state
24     Bid memory newBid;
25     newBid.from = msg.sender;
26     newBid.amount = ethAmountSent;
27     accepted[_id].push(newBid);
28     emit BidSuccess(msg.sender, _id);
29 }
30
31 function cancelAuction(uint _id) public isOwner(_id) {
32     Auction memory myAuction = auctions[_id];
33     uint bidsLen = accepted[_id].length;
34
35     // refund the last bid, if prior bids exist
36     if(bidsLen > 0) {
37         Bid memory lastBid = accepted[_id][bidsLen - 1];
38         if(!lastBid.from.send(lastBid.amount)) revert();
39     }
40     auctions[_id].active = false;
41     emit AuctionCanceled(msg.sender, _id);
42 }
```

**Figure 2: Smart Contract Functions in Auction DApp**

(e.g., the owner is not satisfied with existing bidding prices). However, `cancelAuction()` does not enforce this policy and hence allows owner to cancel an auction at any stage.

***Problem and Challenge.*** It is in fact a challenging problem to automatically apply the exact safety specifications to a smart contract function, since it requires a safety verifier to recognize the semantics of this function and its variables. For instance, to detect the safety problem in the motivating example, a verifier must be able to understand that the `cancelAuction()` function is used to *"terminate a bidding process"*, and correlate the variables `auctions[_id]` and `accepted[_id]` to *"this auction"* and *"prior bids accepted in this auction"*, respectively. Only then can the verifier discover the violation of the aforementioned fairness policy, since the function allows to deactivate an auction (`auctions[_id].active = false`) when a valid bid exists (`bidsLength != 0`).

Prior work that verifies contract safety (e.g, VerX [45]) infers these semantics via manually reading symbols such as function and variable names from source code. However, this approach is neither effective nor robust. To address this issue, we propose to automatically recover semantics of smart contract functions and key variables. Particularly, in our case, we aim to automatically *(1)* discover the high-level logic of the two functions and therefore correctly apply bidding policies to `bidOnAuction()` and canceling

**Table 1: Variable Semantics to be Recovered**

| Variable | Domain | Semantics |
|---|---|---|
| `msg.value` | `bidOnAuction` | bid price |
| `msg.sender` | `bidOnAuction` | bidder |
| `tempAmount` | `bidOnAuction` | last bid price |
| `auctions[_id]` | contract-wide | this auction |
| `auctions[_id].active` | contract-wide | this auction's state (true or false) |
| `myAuction.deadline` | contract-wide | timestamp |
| `myAuction.owner` | contract-wide | a special participant |
| `accepted[_id]` | contract-wide | previously accepted bids in this auction |

rules to the other; *(2)* associate both function-wide and contract-wide variables to auction-specific concepts as shown in Table 1.

## 2.2 Threat Model & Approach Overview

***Threat Model.*** Our trusted computing base consists of *(a)* blockchains, *(b)* smart contract runtime, *(c)* web servers running DApp services, *(d)* client-side browsers, and *(e)* communications between services and browsers. We consider blockchain algorithms and infrastructure to be trustworthy. Any attacks that aim to exploit vulnerabilities in consensus mechanism [24], mining [22] or peer-to-peer network [27] are beyond the scope of this work. We do not consider lower-level software attacks, that affect the integrity of operating systems, smart contract runtime or user browsers; we do not consider network-level attacks that attempt to interrupt or intercept traffic. We consider application-level vulnerabilities, which lie in the implementation of back-end smart contracts, the presentation of front-end interfaces and the interaction between them. First, smart contract code may be incorrect, insecure, unsafe or unfair due to mistakes made by developers. These mistakes can be exploited by malicious users to deliberately misuse benign contracts and cause financial losses to other users or contract owners. Second, DApp web interfaces can be benign yet incomplete or inaccurate. There may exist a gap between descriptive texts on GUI components and internal logic in corresponding contract functions.

***Approach Overview.*** Based upon the threat model, we have developed a system, VetSC, that can automatically perform safety vetting of smart contract in DApps via autonomously interpreting contract semantics and precisely applying corresponding safety specifications. Figure 3 depicts the architecture of VetSC, which consists of three steps. The inputs of VetSC are DApps and a set of manually-crafted high-level business-logic and safety specifications; the outputs are risk reports. A one-time manual effort is needed to create these specs for each app category. We thus expect the specs of VetSC to be written by domain experts, who have good understanding of business logics and safety problems.

(1) **Model Extraction.** Given a DApp, we first extract the business model of every contract function. In particular, starting from key transaction properties in each function, we perform static control-flow and dataflow analyses to build *business model graphs* that represent the function's semantic model.
(2) **Semantics Recovery.** Next, we recover the semantics via UI-guided model checking. More concretely, we analyze the DApp's web code to correlate each contract function to a front-end widget that triggers this function. We then apply natural language processing techniques to the texts on the widget, in order to infer the business logic the widget describes. Guided by the extracted textual semantics, we can thus selectively check *business model graphs* against only relevant logic specs in the input specification set. If a major inconsistency is detected, we will report this issue and terminate our analysis.
(3) **Safety Vetting.** If the extracted graphs are consistent with the described semantics, we can then determine the business logic of the function. Because our model checking has correlated abstract concepts in specs to concrete variable names in specific smart contract implementations, we can leverage this mapping to automatically concretize high-level safety rules and check target functions against custom safety policies.

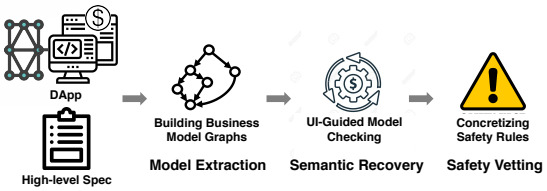


**Figure 3: Architecture Overview of VetSC**

# 3 MODEL EXTRACTION

## 3.1 Smart Contract Semantics Model

We first define a semantics model that represents high-level business logic of smart contracts. This model must abstract away irrelevant low-level implementation details, such as auxiliary data structures, variable declarations or assignments, while capturing intrinsic contract properties and their dependencies.

***Key Factors.*** To this end, we consider the following key factors:

- **Transaction Property.** Transaction-level properties, such as sender of the transaction message (*msg.sender*), are essential metadata to all external smart contract functions. Furthermore, these properties can carry high-level semantics that are tied with unique business logics. For instance, *msg.sender* represents a *bidder* in an auction bidding process but a *voter* in an election. A full list of transaction properties is defined in Solidity documentation [7].
- **Global Variables.** The state of a smart contract is stored as global variables in a special *storage* region. These global variables can be used to maintain persistent and crucial states across all transactions, and any actions taken on them correspond to significant changes to contract status.
- **Dataflow.** The data dependencies between these invariant contract states and transaction-level properties reliably indicate behavior-level operations (e.g., deposit, withdraw).
- **Condition Check.** Similarly, the condition checks that compare these special properties with certain local variables also reflect critical business logic. Our motivating example has demonstrated such a case where a new bid price must be checked to see if it is higher than the current highest bid, before it can be accepted.
- **Cryptocurrency/Token Transfer.** Smart contracts enforce contract-like business logic, which often involves critical funds transfers.

***Formal Definition.*** We then formally define a semantics model as a *Business Model Graph* (`BMG`). In general, A `BMG` depicts the causal (data or control) dependencies among the *essential statements*, where "essential statements" either directly contain transaction properties or global variables, or have data or control dependence on other essential statements.

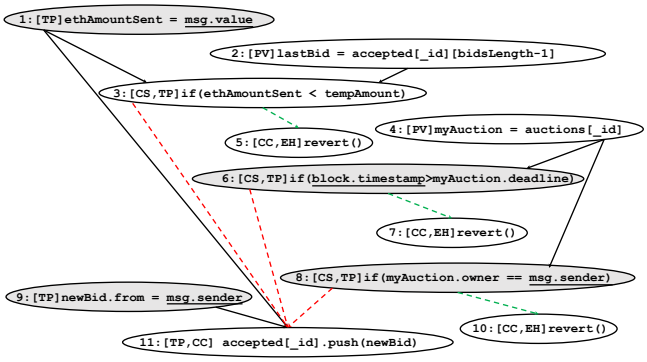


**Figure 4: BMG of the `bidOnAuction()` Function**

**Definition 1.** A *Business Model Graph* is a directed graph $G = (V, E, \alpha, \beta)$ over a set of statements $\Sigma$ and a set of relations $R$, where:

• The set of vertices $V$ corresponds to the statements in $\Sigma$;

• The set of edges $E \subseteq V \times V$ corresponds to the *causal dependencies* between statements.

• The labeling function $\alpha : V \rightarrow \Sigma$ associates nodes with the labels of corresponding statements. Each label consists of three elements: an ID, a set of attributes, and a smart contract statement. There exist eight types of attributes: 1) transaction property [TP], 2) global variable [GV], 3) conditional statement [CS], 4) variables in predicates [PV], 5) conditional clause [CC], 6) external call [EC], 7) call parameter [CP] and 8) exception handling [EH]. Types of attributes are identified via opcodes. For instance, [TP] *msg.sender* can be recognized by opcode 0x33 (`CALLER`).

• The labeling function $\beta : E \rightarrow R$ associates edges with the labels of corresponding relations, which can be 1) data dependency, 2) "true" branch and 3) "false" branch.

`BMG` ***of Motivating Example.*** Figure 4 demonstrates the `BMG` of the `bidOnAuction()` function. Here, statements are presented in source code for readability. The implementation details of bytecode-level analysis will be elaborated later in this section.

In this graph, we have captured four "USE" statements (Vertex #1,6,8,9) of three transaction properties, *msg.value*, *msg.sender* and *block.timestamp*. *msg.value* represents the new bid price, which is saved to a local variable `ethAmountSent` and further checked against `tempAmount` (Vertex #3). In a legitimate bidding process, such a comparison validates the bid price. Thus, `tempAmount` should be corresponding to the current highest bid. Tracing back, we understand that the variable `tempAmount` in the predicate originates from the last element of an array `accepted[_id]` (Vertex #2), and may infer that this array maintains the record of previous bids. All these data dependencies are depicted using solid lines, where arrows point to the destinations. If the given bid price `ethAmountSent` is not higher, the function will raise an exception and revert to the state prior to its invocation (Vertex #5). Otherwise, a write operation on the *storage* will eventually happen (Vertex #11). The control dependencies between the IF statement and the two clauses are depicted as a dotted line, while the green line means the true branch, and then red line indicates the false one. The other two conditional statements we have captured (Vertex #6,8) ensure the auction has not expired and bidder is not the merchandise owner, respectively. Similarly, their conditional clauses lead to either a *storage* write or a `revert()`. Their variables, which are compared against these properties in the predicates, originate from an Auction instance (Vertex #4). Our graph also illustrates the dataflow from *msg.value* to `newBid`, as well as the data dependencies between *msg.sender* and `newBid.from`. At last, a valid `newBid` instance is pushed to the array `accepted[_id]` (Vertex #11), which is stored in the *storage* region and can thus be accessed by other contract functions such as `cancelAuction()`.

## 3.2 Model Construction

***Algorithm.*** VetSC creates one `BMG` for each external function called by the front-end, via context-sensitive, flow-sensitive interprocedural static analysis. Algorithm 1 depicts how we build a `BMG`. The algorithm takes as input a smart contract function `SC`, and produces the corresponding `BMG`. To facilitate subsequent dataflow analysis, our algorithm begins with a custom points-to analysis on memory operations that access global variables (details will be discussed later). Then, it collects an initial set $\mathbb{INIT}$ of essential statements (statements with transaction properties or global variables) in `SC`.

For every definition statement `stmt` in this set, we first conduct context-sensitive, flow-sensitive inter-procedural dataflow analysis to find its $\mathbb{SINK}$ set; for each `sink`, if $<$ `stmt`, `sink` $>$ is a new edge, we add it into the graph. Next, we check the type of every `sink` to see whether the statement is a condition check (Ln.10), a cryptocurrency/token transfer (Ln.17) or a global variable write operation (Ln.19). Depending on its type, we obtain *(a)* predicate variables in a condition statement, *(b)* address and value variables in a transfer, or *(c)* offset and value variables in a global memory access, and save them into $\mathbb{SVAR}$. Then, we perform a backward dataflow analysis on these variables to identify their sources, and add new edge $<$ `src`, `svar` $>$ to `BMG`.

In addition to data dependencies, we also compute control dependencies (Ln.10). In particular, we perform inter-procedural control-flow analysis to locate all the statements in the branches $\mathbb{BRANCH}$ that are dependent on the conditions. Within $\mathbb{BRANCH}$, we specifically search for state-changing operations (Ln.13), including global variable accesses and exception routines (i.e., `revert()`). Our algorithm discovers such operations as well as their triggering conditions ("true" or "false") and stores each pair (`op`, `cond`) into a set $\mathbb{OP}$. A control-flow edge between `sink` and `op` under condition `cond` will be inserted into the graph.

***Implementation and Special Considerations.*** Our analysis is performed directly on Ethereum virtual machine (EVM) bytecode. We have implemented our graph construction algorithm on top of a static analyzer Octopus [3], which provides basic analysis capabilities, such as function identification, control-flow analysis and static single assignment (SSA) format transformation. Particularly, our dataflow analysis is conservative and does not differentiate array elements. Our inter-procedural analysis considers a depth of 2. Note that, the improvement of program analysis is orthogonal to our focus, which is to interpret program semantics. VetSC can potentially leverage any advanced static analysis tools to improve precision. Nevertheless, our current analysis can already extract relatively precise and complete graphs.

*Transaction Properties Identification.* To locate the initial sources for our dataflow analysis, we must automatically identify transaction properties in smart contract bytecode. To this end, we have studied the EVM documentation [5] and found that the usages of transaction properties are implemented as special instructions. For

**Algorithm 1** Graph Construction

```
1:  procedure BuildBMG(SC)
2:      BMG ← ∅
3:      PointsToAnalysisForGlobalVariables(SC)
4:      INIT ← InitialEssentialStmt(SC)
5:      for ∀stmt ∈ INIT do
6:          SINK ← ForwardDataFlowAnalysis(stmt)
7:          for ∀sink ∈ SINK do
8:              BMG ← BMG ∪ < stmt, sink >
9:              SVAR ← ∅
10:             if sink is ConditionCheck then
11:                 SVAR ← GetPredVars(sink)
12:                 BRANCH ← ControlDepAnalysis(sink)
13:                 OP ← FindStorageOpsOrExceptions(BRANCH)
14:                 for (op, cond) ∈ OP do
15:                     BMG ← BMG ∪ < sink, op >cond
16:                 end for
17:             else if sink is Transfer then
18:                 SVAR ← GetAddr(sink) ∪ GetValue(sink)
19:             else if sink is GlobalVariableWrite then
20:                 SVAR ← GetOffset(sink) ∪ GetValue(sink)
21:             end if
22:             for ∀svar ∈ SVAR do
23:                 src ← BackwardDataflowAnalysis(svar)
24:                 BMG ← BMG ∪ < src, svar >
25:             end for
26:         end for
27:     end for
28:     return BMG
29: end procedure
```

example, *msg.sender* can only be accessed by an EVM instruction `CALLER`, and the `CALLVALUE` instruction is used to retrieve *msg.value*. Since there are only limited numbers of transaction properties and the instructions are fixed, we can thus maintain a pre-defined instruction list to discover these properties in bytecode programs.

*Memory Aliasing.* The unique memory model of EVM is a major obstacle for precise dataflow analysis. EVM implements two types of global memory regions *memory* and *storage*. The former is commonly used to hold temporary data during a transaction, and the latter is designed to maintain persistent contract states. Frequent data exchanges through *memory* and *storage* via subsequent stores and loads make dataflow obscure. For instance, Figure 2 first writes *msg.sender* to a *memory* variable `newBid` (Ln.25), and then reads from this *memory* region and stores the obtained data to the global *storage* `accepted[_id]` (Ln.27). This is a typical alias problem, as we need to figure out that the *memory* address being written to is identical to the one later being read. However, finding memory aliases can be nontrivial because EVM implements an offset addressing mode, and such an offset is often computed using complex arithmetic or even hash functions – e.g., at the bytecode level, a *storage* operation, such as `SLOAD` or `SSTORE`, can access data at a computed offset `(x+30)` or a hashed one `SHA3(0,40)`.

To address this challenge, we develop a custom points-to analysis for *store* and *load* so as to explicitly connect corresponding operations. At a high level, we use the structure of computation trees as a signature to represent a memory offset. Particularly, we first scan the whole DApp to identify *memory* and *storage* access instructions. Then, we perform backward dataflow analysis on each memory offset to generate an arithmetic formula, such as `(SHA3(0,20)+x)/y`, in the form of a tree. Such a tree denotes the calculation process of this offset. Hence, we can match a memory store with a load based upon the graph edit distance between their offset calculation trees. An exact match indicates a dataflow from the store to the load. To further improve accuracy, we perform simple arithmetic optimization to correlate expressions (e.g., "3+5") with values (e.g., "8"). For instance, in the motivating example, a `MSTORE` stores *msg.sender* to *memory* `newBid` at offset "0" (Ln.25) and a `SHA3` (an opcode for load) later loads data from the same offset (Ln.27). Thus, we can establish the data dependency between these two operations.

Notice that our matching technique can potentially lead to false negatives in theory. However, because an entire contract is compiled using only one version of Solidity compiler, which calculates offsets for the same memory location in the same manner, we did not observe inaccuracy in practice.

**Table 2: Mined Keywords for Smart Contract Functions**

| Contract-level Keywords | Function-level Keywords |
|---|---|
| Voting | *vote* |
| Auction | *bid, cancel, finalize* |
| Wallet | *deposit, withdraw, transfer* |
| Gambling | *play, buy, refund, draw* |
| Trading | *buy, sell* |
| Crowdsale | *buy/invest, close, refund* |

# 4 SEMANTIC RECOVERY & SAFETY CHECK

To automatically discover the semantics of our extracted graph model, we develop a UI-guided model checking technique. Specifically, we collect UI information relevant to each individual smart contract function, and then use model checking to match the business logic inferred from the user interface with the `BMG` of the corresponding function. A successful match reveals semantics of a contract function and its key variables. Hence, we can further automatically select relevant safety policies to examine this function.

## 4.1 UI Semantic Inference

***Identifying Contract Function related UI Text.*** To find relevant UI information, we use fuzzing to iteratively explore each DApp widget while monitoring the smart contract function it invokes, to identify UI components corresponding to each back-end contract function. To gracefully trigger widgets consecutively without unexpected termination, we leverage the *smart input generation* technique proposed by SMV-Hunter [55], which uses input validation code to determine the valid input types of editable widgets. Our implementation is based upon the MetaMask [2] browser extension. We hook the creation point of a "payment page", which is rendered immediately prior to any `eth_call` – the JSON RPC that eventually makes smart contract calls. We can thus retrieve the contract address and the target function from each `eth_call`, and therefore build the connection between front-end widgets and internal functions.

Then, we collect descriptive texts from these widgets. In our motivating example, we can directly obtain *"Bid Now"* and *"Cancel Auction"* from the two buttons. Nevertheless, the texts on widgets alone may not be sufficient to infer widget semantics; the context of widgets also matters. For example, the button *"Buy Now"* in an auction app indicates the action to place a bid, while a button with the same text in a gambling app means "to place a bet".

To collect contextual texts of a widget, we follow a prior work [11] that inspects application user interfaces. Particularly, we leverage an automatic testing tool Selenium [9] to extract and serialize a DOM tree and use the neighboring textual elements within a fixed window size to represent context (empirically set to 5).

**Table 3: Specifications for Motivating Example**

| | Function | Spec Type | Formal Spec | Explanation |
|---|---|---|---|---|
| Auction | Bidding | Essential#1 | $\Box(current\_bid > highest\_bid \rightarrow \Diamond(highest\_bid := current\_bid \wedge highest\_bidder := current\_bidder))$ | Accept only higher new bid |
| | | Safety#1 | $\Box(current\_time > deadline \rightarrow \Diamond(execution\_state := revert))$ | No bidding on expired auction |
| | | Safety#2 | $\Box(auction.active == false \rightarrow \Diamond(execution\_state := revert))$ | No bidding on inactive auction |
| | | Safety#3 | $\Box current\_bidder == auction.owner \rightarrow \Diamond(execution\_state := revert))$ | Auction owner cannot bid |
| | Cancel | Essential#1 | $auction.active := false$ | Auction state becomes inactive |
| | | Safety#1 | $\Box(requester != auction.owner \rightarrow \Diamond(execution\_state := revert))$ | Only owner can cancel an auction |
| | | Safety#2 | $\Box(highest\_bidder != null \rightarrow \Diamond(execution\_state := revert))$ | Cannot cancel when a valid bid exist |

***Inferring UI Semantics via NLP.*** Once UI texts with their contexts are collected, we can infer their semantics via NLP. If two 'Buy Now' buttons process different logics, we can still handle them correctly as long as the two logics are well-defined in the business model, and the contexts of the two buttons are different. The insight is that the texts and contexts of the buttons should contain enough information for end users (and our NLP) to differentiate.

To handle fragmented widget texts, we infer UI semantics based on words rather than sentences; to build semantic templates for Dapps, we mine high frequency words that are used in popular DApps from real-world markets. Particularly, we have established two levels of templates: contract-level and function-level. We have identified 6 contract-level keywords and 16 function-level keywords as shown in Table 2. More concretely, we first clean the obtained texts via removing less significant components (e.g., preposition, numeric words) using POS Tagger [4]. Then, we adopt a two-layer matching process to identify the best match between the extracted widget text and our identified keywords for *(1)* contracts and *(2)* functions. To do so, we embed each keyword into a representation vector $S = [s_1, s_2, ..., s_k]$ using a pre-trained model GloVe [30], to avoid any overfitting problem. For each UI widget, we generate two semantic representations $S_{text}$ and $S_{context}$ for its text and context respectively. *Layer 1:* We infer the business logic of the whole DApp by calculating the distance between $S_{context}$ and the embeddings of all business categories, and consider the closest one as the inferred business logic. *Layer 2:* Once the business logic is determined, we perform the second matching to compare $S_{text}$ with the embeddings of function keywords that belong to the matched category to determine the function-level semantics of the UI widget. Notice that if no comparison yields a score higher than our 50% threshold, the widget will not belong to any of the six categories.

In the motivating example, the contexts of "BID NOW" and "CANCEL AUCTION" are a set of words: *{auction name, auction creator, starting price, current bid, bidder number, last bidder, expire, auction state}*. Our UI semantics inference identifies the business logic *Auction* with a similarity score of 93%, and further correlates "BID NOW" to *bidOnAuction* and "CANCEL AUCTION" to *cancelAuction* with similarity scores of 86% and 97%, respectively.

## 4.2 Model Checking

Once we have extracted textual semantics from widget text, we then check whether the UI described business logic matches that of the contract code. If so, we can further recover the semantics of individual contract functions and variables. Otherwise, we will report the UI-logic inconsistency. To enable such a model checking, we have modeled the semantics of contract code using our derived `BMGs`, and we will next craft business logic and safety specifications for the semantics inferred from text.

***Specification.*** We first formally define our specifications for model checking, which are expressed using LTL. Note that while we perform business model checking and safety vetting separately, the same formalization of specifications are applied to both.

**Definition 2.** Let $P$ be a set of atomic logical proposition symbols about the system $\{p_1, p_2, ...p_{|A|}\}$, e.g., the *bidder* is the *owner* of product, and let $\Sigma = 2^A$ be a finite alphabet composed of these propositions. Then, a set of **LTL-based Specifications** is inductively defined by the grammar:

$$\varphi ::= true \mid p \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \, U \, \varphi_2 \tag{1}$$

, where $\neg$ and $\wedge$ denote negation and logical AND operators; $\varphi_1 \, U \, \varphi_2$ indicates that $\varphi_1$ remains true until $\varphi_2$ becomes true; $\bigcirc\varphi$ means $\varphi$ is true in the next step. In addition, we also use the following redundant notations: $\varphi_1 \vee \varphi_2$ instead of $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2$ instead of $\neg(\varphi_1 \wedge \neg\varphi_2)$, the *eventually* operator $\Diamond\varphi$ instead of $(true \, U \, \varphi)$, and the *always* operator $\Box\varphi$ instead of $\neg\Diamond\neg\varphi$.

We have crafted two types of specifications based on LTL: 1) Essential Logic $\phi_E$, which specifies the basic – yet unique and intrinsic – workflow of business logics, and 2) Safety Rule $\phi_S$ that further specifies the safety constraints for particular business logics.

Table 3 exemplifies the specifications we have defined for the motivating example. For the *Bidding* function, our essential rule captures the key logic of bidding process, which is a comparison between a given bid price *current_bid* and the current highest bid *highest_bid*. Only when the former is greater, this bid becomes the new *highest_bid* and the bidder *current_bidder* becomes the *highest_bidder*. In the meantime, our first safety rules dictates that the bidding transaction should happen before the auction deadline. The second rule requires that the auction be still active. These two rules guarantee the liveness of an auction. The third rule enforces the legal policy for "without reserve" auction, which prevents owners of auctioned merchandises from bidding their own products, avoiding potential price manipulations. Similarly, our essential rule for the *CancelAuction* function specifies that if the current auction exits normally (i.e., no revert), the auction will eventually be deactivated. Our safety rules require that when canceling an auction, 1) the *requester* must be the Auction owner; 2) no valid bid has been received and the highest bid is still null. Notice that we do not claim our specifications can cover all applications. Generating a complete set of logic and safety specifications is a challenging task on its own and also orthogonal to the focus of this work. We expect impactful and comprehensive specs to be created by domain experts.

***A Model Checking Problem.*** We formalize semantic recovery and safety verification as a model checking problem. Specifically, we follow SABOT's approach [40] to check extracted contract models against essential logic specifications. If a match is identified, high-level transaction concepts in the specifications (e.g., `current_bidder`) will then be mapped to the concrete variables in contract code (e.g., `msg.sender`), revealing their semantics. With such a mapping, we then automatically apply a high-level safety specification (e.g., $\Box current\_bidder == auction.owner \rightarrow$

$\Diamond(execution\_state = revert)$) to verifying the low-level code where `current_bidder` is represented as `msg.sender`. At a high level, we aim to establish a mapping between key variables in BMG and the atomic propositions in our predefined specifications.

**Definition 3.** Our goal to find a unique mapping $\mu$ is defined by:

$$\mu = P_{\phi_E} \rightarrow GV \text{ such that } M_{\text{BMG}} \models \mu/\phi_E, where: \quad (2)$$

- $P_{\phi_E}$ is a set containing all the variables $p$ used in propositions in essential specifications $\phi_E$;
- $GV$ is a set containing all the global variables in BMG;
- $M_{\text{BMG}}$ is the transition system that directly presents BMG;
- $M_{\text{BMG}} \models \mu/\phi_E$ means that the specification $\phi_E$ holds over the transition system $M_{\text{BMG}}$ under the unique mapping $\mu$, when the variables $P_\phi$ are mapped to global variables $GV$.

Note that, an entire business logic may actually consist of multiple individual functions. For instance, an auction logic has both `bidOnAuction()` and `cancelAuction` functions. Consequently, our model checking must find a global mapping between all function models in a DApp and all the corresponding specifications.

## 4.3 Semantic Recovery and Safety Vetting

***Transforming* BMG*s* *to NuSMV Models.*** Our implementation is based upon a state-of-the-art model checker NuSMV [6]. We convert BMGs to *identifiers* and *transitions* in the NuSMV model.

In Figure 4, we consider three transaction properties `msg.value`, `block.timestamp` and `msg.sender`, as well as three global variables `newBid.from`, `newBid.amount` and `accepted[_id]-[bidsLength-1].amount` to be *identifiers*. We initialize them with random values. In addition, we also define two other *identifiers* that represent execution context and state: *function* and *execution_state*. The former indicates the name of the contract function being executed. Here, it is either `bidOnAuction` or `cancelAuction`. The latter preserves the status and can be either `normal` or `exception`.

Next, we define *transitions* for each identifier. A *transition* indicates the *condition*, in which an identifier can change, and the resulting *value* due to such a change. Thus, TP identifiers are not associated with *transitions* because their values do not depend on any inner conditions. In contrast, the precondition of a GV-related transition is its hosting function plus its immediate conditional statement, and the value is updated by corresponding store operations. In Figure 4, the transition in Vertex #11 happens when the conditions in Vertex #3, #6 and #8 are all satisfied, and the resulting values originate from Vertex #1 and #9 respectively.

***Recovering Function and Variable Semantics.*** Once we have generated the NuSMV models, we use model checking to recover function and variable semantics. Particularly, starting with one function model and its corresponding essential specification $\phi_E$, we first create a random mapping between the global variables in this model and the high-level concepts in $\phi_E$. Then, we update $\phi_E$ using these global variables, and use NuSMV to check if the updated specification $\mu/\phi_E$ still holds for the model. If so, we further check the next function model. Otherwise, we create another random mapping and continue the test in an iterative manner. Only when we can find a mapping for each individual function model, we conclude that smart contract logic is consistent with described semantics. For the motivating example, our model checker believes the described logic (i.e., auction) matches the smart contract implementation. Table 4 shows our discovered global mapping between contract variables and high-level concepts. If we cannot find such a mapping, we will raise an alarm to indicate the detected inconsistency.

**Table 4: Mapped Key Variables for Motivating Example**

| Function/Domain | Smart Contract Variable | Spec Concept |
|---|---|---|
| bidOnAction | `msg.value` | *current_bid* |
| bidOnAction | `msg.sender` | *current_bidder* |
| bidOnAction | `newBid.amount` | *highest_bid* |
| bidOnAction | `newBid.from` | *highest_bidder* |
| cancelAction | `msg.sender` | *requester* |
| contract-wide | `accepted[_id][bidsLength-1].amount` | *highest_bid* |
| contract-wide | `accepted[_id][bidsLength-1].from` | *highest_bidder* |
| contract-wide | `accepted[_id].active` | *auction.active* |

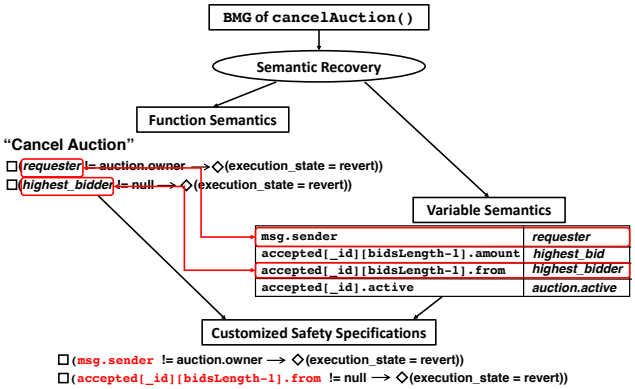


**Figure 5: Generating Customized Specs**

***Detecting Safety Violations.*** When smart contract code conforms to textual semantics of corresponding widgets, we further check its function models against our safety rules. The goal is to automatically generate and apply safety specifications that are customized to a target smart contract function. As depicted in Figure 5, the prior semantic recovery generates two levels of information that can facilitate automated safety checking: *(1)* function semantics and *(2)* variable semantics. The former informs us which set of safety policies need to be applied to a specific function; the latter tells us how to customize them. For instance, because we have successfully matched the function model of `cancelAuction()` with our logic specification for "Cancel Auction", two safety rules in this category will then be checked. Also, since we have associated the *requester* of auction termination to `msg.sender` of `cancelAuction()`, and have correlated *highest_bidder* with the field of a global item `accepted[_id][bidsLength-1].from`, we can rewrite the safety policies by replacing the high-level concepts with concrete variables.

Once we have produced these customized safety rules, we can then verify the BMG of `cancelAuction()` against these rules. The verification is done automatically via standard model checking using the generated model and safety specs. In our example, the verification detects the violation of the second policy, which requires an auction not to be canceled if a valid bid exists.

# 5 EVALUATION

We evaluate the effectiveness of our automated safety vetting, UI analysis and semantic recovery (particularly for large DApps and emerging DeFi apps), as well as runtime efficiency.

## 5.1 Experimental Setup

***Dataset Construction.*** To discover real-world problems while evaluating VetSC with ground-truth data, we collect open-source

DApps projects from GitHub. We need to exclude many token exchange apps (e.g., 69/100 top apps in *dapp.com*) because their business and safety logic are usually **not** implemented in smart contracts but at a custom application level. Nevertheless, collecting "complete" open-source apps for experiments is still a daunting task.

We consider "complete" apps to be self-contained projects. Even top Github projects may only contain codebases pertaining to their own development (e.g., new contract code snippets, added Node.js modules, or individual app components), and may not include complete or correct metadata, dependencies or configurations that are necessary for compiling, loading or executing. To our study, a large portion of top GitHub DApps fails to run due to three major reasons:

*(a)* Back-end smart contract code cannot be built or deployed because of incompatible compiler versions (e.g., compiler features or checks specified by `pragma` in DApps such as *BigBoardzDapp* and *dapp-traceability*) or missing deployment specs (e.g., truffle migration files needed for *FoodVoter* or *Decentralized-Voting-DApp*).

*(b)* Front-end UI code only contains individual core components but does not include important or up-to-date dependencies or entry points. For instance, deprecated Node packages `source-map-resolve` and `source-map-url` are being used (in e.g., *dVoting* and *ether-loto*); precise npm versions are unknown for many projects (e.g., *solidity-auction*, *ethereum-lottery-app*); some apps (e.g., *truffle-vote*) do not provide key HTML interfaces including `index.html`.

*(c)* Even if both back-end functions and front-end widgets are complete, their connections can be misconfigured. For example, in the projects *CoinFlipper-DApp* and *Bank-DApp*, back-end logic cannot be successfully triggered by front-end GUI events.

As a result, these apps are considered incomplete from our evaluation perspective, because VetSC needs to perform dynamic analysis to correlate front-end HTML text and JavaScript code with back-end contract calls at runtime.

**Data Collection.** Nevertheless, we still manage to collect 34 workable DApps. To the best of our knowledge, this is the *largest* dataset to date for safety vetting of DApps. To collect them, we leveraged GitHub engine to search for "best match[ing]" open-source, non-token projects using the keyword "DApp + <Business Model Category>" and retrieved top 100 apps from each of our 6 categories. We then identified 144/600 apps with UI. We further excluded incomplete apps that cannot be set up and eventually obtained 30 workable apps. We also added 2 closed-source apps whose bytecode semantics we can manually understand, and used 2 VerX's crowdsale apps via adding synthesized textual semantics.

**Impact of Collected Apps and Homogeneity of DApps.** Our samples are representative due to their high impacts and the homogeneity of current DApps. Out of the 34 test apps, we can manually verify (via metadata, website information, simple code match) that 11 of them have been deployed on Ethereum Mainnet. We then find the transaction history for the addresses of the corresponding contracts. Our study shows that we have covered several very high-profile DApps. For instance, ~5 million transactions have been processed for the *CryptoKitties* app; 780K transactions have been made for the *etheroll* app. Overall, a majority (over 70%) of our test apps have many active users, leading to at least 2K transactions.

Besides, our study also reveals that there exist many duplicates of the 34 apps and their core functions (e.g., bidding, trading, etc.) both on the Ethereum blockchain and in GitHub projects. Using a simple string matching, we can already discover 1,396 duplicates (507 from blockchain, 889 from GitHub) whose contract code is identical to that of our apps. In fact, prior studies Oyente [39] and Zeus [33] have also made the similar observations where only 19% and 6.7% apps in their datasets, respectively, are unique. Current DApps are highly homogeneous possibly due to the limited types of financial applications (e.g, bidding, trading), and some *de facto* standard implementations of their core logic.

***Environment.*** Our experiments have been conducted on a machine equipped with Intel i7 CPU @ 3.20GHz and 16GB of physical memory. The OS is Ubuntu 18.04 LTS (64bit).

## 5.2 Detection Results

We apply VetSC to the automated safety vetting of these 34 DApps, and Table 5 presents our overall detection results. In general, we have identified 19 unsafe functions. These unsafe functions violate our predefined policies and may incorrectly allow dangerous or unfair actions such as double voting or buying an expired lottery ticket. We have also measured the correctness of our semantic recovery and safety vetting, discussed at the end of this section.

***Safety Violations.*** We have discovered unsafe functions in a variety of financial applications including 2 lottery (gambling) games, 2 auction apps, 9 voting platforms and 1 crowdsale applications.

For instance, #10 *all-for-one.club*, a lottery game that draws on a daily basis, has two unsafe functions `draw` and `buy`. While lottery games are extremely time sensitive, this app fails to set a deadline and check whether a game has already expired when a drawing starts or when players purchase tickets for a specific game. As a result, these two functions violate our safety rules `Lottery-Draw-S2` and `Lottery-Buy-S1`, respectively. Similarly, another lottery app #16 *Lottery-DApp* shares the same safety issues.

Time also matters for a safe and fair implementation of auction applications. Unfortunately, #11 *openberry-ac* does not carefully take time into account and thus incorrectly allows bidding for an already expired auction. In addition to the timing factor, auction games have strict requirements on restricting sellers to arbitrarily cancel an auction with valid bids, as indicated by auction rule [20]. However, the auction app #17 *mastering-ethereum-auction-dapp* does not actually enforce this policy `Auction-Cancel-S2` and thus may cause an unfair consequence.

Likewise, most of the unsafe election apps we have detected also suffer from missing time checks. Furthermore, #12 *create-react-dapp*, a Github project that demonstrates how to establish a voting platform, may even cause a serious *double voting* problem because it does not check if a voter has already submitted a ballot.

VetSC detects safety risks in crowdsale apps as well. For example, #33 *Overview* contains a problem that mistakenly accepts new investment when a crowdsale has expired and has already refunded existing investors. This vulnerability has also been confirmed by the state-of-the-art VerX [45] tool. However, it is worth noting that although VerX can achieve the same detection results for crowdsale apps #33 and #34 where source code is available, it requires significant manual efforts to create custom safety policies for different implementations of same logic. We show in Section 5.3 that how VetSC can automate this process to facilitate safety verification.

***UI-Logic Consistency.*** Table 5 also shows the textual descriptions on the widgets that are associated with the unsafe contract

**Table 5: Detection Overview**

| # | Name | Unsafe Func Name | Code Logic | Major Widget Text/Context | UI == Logic? | Safety Issue in Smart Contracts | Violated Policy | Source Analyzability |
|---|---|---|---|---|---|---|---|---|
| 1 | cryptoatoms.org | - | - | - | Yes | - | - | Yes |
| 2 | proofoflove.digital | - | - | - | Yes | - | - | Yes |
| 3 | snailking | - | - | - | Yes | - | - | Yes |
| 4 | cryptominingwar | - | - | - | Yes | - | - | Yes |
| 5 | market.start.solar | - | - | - | Yes | - | - | No (Missing Source) |
| 6 | etheroll | - | - | - | Yes | - | - | No (Inlined Bytecode) |
| 7 | cryptokitties | `bid()` | Auction-Bid | *"buy"* | Ambiguity | N/A | N/A | Yes |
| 8 | hyperdragons | - | - | - | Yes | - | - | No (Missing Source) |
| 9 | dice2.win | - | - | - | Yes | - | - | No (Inlined Bytecode) |
| 10 | all-for-one.club | `drawNow()` | Lottery-Draw | *"Draw"* | Yes | Drawing for an expired lottery | Lottery-Draw-S2 | No (Inlined Bytecode) |
| | | `play()` | Lottery-Buy | *"pay 1 ETH"* | Yes | Buying an expired ticket | Lottery-Buy-S1 | No (Inlined Bytecode) |
| 11 | openberry-ac | `placeBid()` | Auction-Bid | *"place BID"* | Yes | Bidding for an expired auction | Auction-Bid-S1 | Yes |
| | | `finalizeAuction()` | Auction-Close | *"handle Finalize"* | Yes | Closing a non-expired auction | Auction-Close-S1 | Yes |
| | | | | | | Closing an active auction | Auction-Close-S2 | |
| 12 | create-react-dapp | `voteForCandidate()` | Voting-Vote | *"vote Rama/Nick/Jose"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| | | | | | | Double voting | Voting-Vote-S2 | |
| 13 | ethereum-voting | `vote()` | Voting-Vote | *"Vote"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 14 | ethereum-wallet | - | - | - | Yes | - | - | Yes |
| 15 | heiswap.exchange | - | - | - | Yes | - | - | No (Inlined Bytecode) |
| 16 | Lottery-DApp | `makeGuess()` | Lottery-Buy | *"Buy", "Lottery"* | Yes | Buying an expired ticket | Lottery-Buy-S1 | Yes |
| | | `closeGame()` | Lottery-Draw | *"Close Game", "Lottery"* | Yes | Drawing for an expired lottery | Lottery-Draw-S2 | Yes |
| 17 | mastering-e-a-d | `cancelAuction()` | Auction-Cancel | *"CANCEL AUCTION"* | Yes | Seller cancel after bidding starts | Auction-Cancel-S2 | Yes |
| 18 | multisender.app | - | - | - | Yes | - | - | Yes |
| 19 | note_dapp | - | - | - | Yes | - | - | Yes |
| 20 | metacoin | - | - | - | Yes | - | - | Yes |
| 21 | simple-vote | `vote()` | N/A | *"Start a vote"* | No Impl. | N/A | N/A | Yes |
| 22 | truffle-voting | `vote()` | Voting-Vote | *"Approve/Against/Abstain"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 23 | Gnosis Safe | - | - | - | Yes | - | - | Yes |
| 24 | vote-dapp | - | - | - | Yes | - | - | Yes |
| 25 | EVotingDApp | - | - | - | Yes | - | - | Yes |
| 26 | Election | `vote()` | Voting-Vote | *"Vote"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 27 | Election-DAPP | `vote()` | Voting-Vote | *"Approve/Against/Abstence"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 28 | Vote | `vote()` | Voting-Vote | *"Submit"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 29 | VotingDapp | `vote()` | Voting-Vote | *"Vote"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 30 | VoteDapp | `vote()` | Voting-Vote | *"Vote"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 31 | voting-DApp | `vote()` | Voting-Vote | *"Vote"* | Yes | Voting for an expired election | Voting-Vote-S1 | Yes |
| 32 | VoteMe | - | - | - | Yes | - | - | Yes |
| 33 | Overview | `invest()` | CS-Invest | *"Buy tokens", "Crowdsale"* | Yes | Invest an expired crowdsale | CS-Invest-S2 | Yes |
| 34 | Crowdsale | - | - | - | Yes | - | - | Yes |

functions. In fact, a large majority of the widget texts can faithfully reflect the underlying code logic and therefore can help us precisely identify the function-level semantics of corresponding smart contracts. *This overall consistency between DApp UI and backend logic justifies the feasibility of our proposed approach*, which, to our knowledge, is the first attempt to apply such auxiliary information to the automated interpretation of low-level, obscure contract code.

Nevertheless, we do observe inconsistencies in two DApps #7 *cryptokitties* and #21 *simple-vote*. In the famous game #7 *cryptokitties*, a widget which is described by its text and context as *"buy"* – and thus interpreted as "trading" by VetSC – is actually associated with a contract function that, however, exercises a bidding process. This could cause an ambiguity to human readers, although maybe at a moderate level. #21 *simple-vote* contains a more obvious mismatch where its described *"vote"* logic is not actually implemented in the corresponding contract code.

Again, due to the actual presence of inconsistent UI texts, we do not assume the faithfulness of descriptive texts when interpreting relevant contract logic. Instead, VetSC can directly report such a discrepancy via our model checking. VetSC will proceed to safety verification only when the UI-logic consistency is verified.

***Source Code-Level Analyzability.*** Quite a few DApps under study do not have clear source code. Manual semantic interpretation, used by previous work such as VerX [45] and Zeus [33], thus cannot apply. Specifically, some proprietary contract programs have already been deployed as bytecode to the blockchain and source code is completely withheld by their developers. Additionally, the source code of other contracts, although disclosed to the public, has been "obfuscated" through inlined Ethereum bytecode. Figure 6 depicts such an example. In the source code of *all-in-one.club* lottery app, a helper function invoked by `draw()` to generate random numbers uses bytecode instruction `mload`, `mstore` to directly access data from specified addresses, probably for the sake of runtime efficiency. However, this can further affect the readability of smart contract code and may make manual analysis in prior work extremely hard.

```
function appendInt(buffer memory buf, uint data, uint len)
    internal constant returns(buffer memory) {
    ...
    assembly {
        let bufptr := mload(buf)
        let buflen := mload(bufptr)
        let dest := add(add(bufptr, buflen), len)
        mstore(dest, or(and(mload(dest), not(mask)), data))
        mstore(bufptr, add(buflen, len))
    }
    return buf;
}
```

**Figure 6: Inlined Bytecode in *all-in-one.club* Contract Code**

***Correctness.*** While VetSC in general can correctly discover aforementioned safety issues, we do observe false positives in both our semantic recovery (2/49 functions associated with UI texts) and safety check (3/38 safety related functions). In particular, VetSC mistakenly detects semantic inconsistencies in #8 *hyperdragons* and

**Table 6: VetSC vs. VerX**

| Analysis Step | VetSC | VerX |
|---|---|---|
| *(a)* High-level Specs | One-Time Manual Effort | None |
| *(b)* Semantic Recovery | Automated (UI-Guided) | Manual Effort for Each Func |
| *(c)* Specs Customization | Automated | Manual Effort for Each Func |
| *(d)* Safety Verification | Automated | Automated |

#20 *metacoin*, in terms of our policies `Trading-Buy-Essential#1` and `Wallet-Withdraw-Essential#2`, respectively. These false alarms are caused due to the indirect "fund flows". While the contracts in fact conform to our essential rules and *send* funds to a payee, they do not implement such a transaction directly using the `send()` API. Instead, they will first add funds into payees' virtual accounts, and transfer the funds while reducing virtual balances later. Our static analysis does not track such fund flows that perform arithmetic on virtual balances, and thus miss the indirect fund transfer.

VetSC has also caused three false positives for apps #10, #15 and #22, when checking the `Lottery-Draw-Safety#2` policy (no drawing before deadline), `Wallet-Withdraw-Safety#1` rule (balance check), and `Voting-Vote-Safety#2` (no double-voting), respectively. DApp #10 actually checks the end time of a lottery game but in an implicit manner. In lieu of testing the current time against a final deadline, the contract checks it against each end time of an individual "round". VetSC does not correlate these "sub-deadlines" with the final one and thus does not recognize such a time check.

Apps #15 and #22 both contain very complex arithmetic expressions in their memory addressing and thus lead to expensive graph matching. VetSC, in practice, places a limit on the graph size to avoid overly expensive graph edit distance calculation, and does not find a match in these cases, and thus results in a broken dataflow.

## 5.3 Comparison with VerX

Despite the fact that manual semantics interpretation used in previous safety verification tools cannot work on apps whose source code is missing or obfuscated, they can still be applied to DApps where contract source code is available. As a result, we would also like to understand the advantage of applying VetSC to these apps, via directly comparing it with a representative verifier, VerX [45].

Ideally, we hope to submit our detected unsafe apps to the VerX service via its public APIs [14]. However, we were not able to successfully launch the tool on these new samples. Consequently, we instead apply VetSC to two publicly available crowdsale apps in VerX paper, *Overview* and *Crowdsale*, to show how VetSC can solve the same problems in a more automated fashion.

Table 6 illustrates the difference between VetSC and VerX. In general, VetSC can automate several key steps that require tedious manual efforts in VerX.

## 5.4 Effectiveness of UI Semantic Inference

**UI Analysis on Random Widgets.** We evaluate our UI semantic inference on 185 manually labeled micro-benchmark widgets, obtained from 37 DApps (14 Trading, 9 Voting, 4 Wallet, 5 Auction and 5 Gambling) and 54 web pages.

150 out of the 185 widgets can trigger smart contract functions. We manually verify that we can correctly infer the semantics for 82.5% of the 150 widgets. In contrast, for the 34 widgets that do not lead to contract calls, such as *"Show More", "Need Help?"*, our experiments show that a large portion (65%) has actually been incorrectly classified as one of these business logics. Note that, this in fact demonstrates the advantage of our approach, which, instead of applying NLP to all widgets (relevant and irrelevant) blindly, focuses particularly on those that can trigger contract functions and thus can improve the inference accuracy.

Further study indicates that our context retrieval is helpful to understanding widget semantics. For instance, Gambling DApps *etheroll.com* and *dice2.win* each have a button to place a bet. While the former names the button *"roll"*, the latter calls it *"bet"*. Although the two buttons contain different texts, we can correctly associate them to the same semantics, due to the interpretation of their contexts. Similarly, a Trading app *proofoflove.digital* charges users cryptocurrency, through a button *"Prove Love"*, in exchange for permanently "engraving" a relationship on an immutable blockchain. Although the widget text is very uncommon comparing to the normal vocabulary of trading, we can still capture its real meaning based on its context. Nevertheless, we notice that our semantic inference is still constrained by the common limitation of NLP and therefore may cause incorrect classification. The major barrier originates from rhetoric and metaphor. For example, the auction DApp *King of the Ether Throne* [1] has a button *"Claim the Throne"* which in fact places a bid but cannot be correctly identified because the phrase highly deviates from common texts used in bidding processes. Notice, however, while such rhetoric cannot be easily interpreted by NLP, it may also cause misunderstanding for human readers and thus lead to the discrepancy between UI description and internal smart contract implementation.

**UI Analysis on Large Apps.** We would further like to understand if VetSC can successfully extract textual semantics from large DApps with complex user interfaces. To this end, we apply our UI analysis to marketplace apps that contain non-trivial web UI for auction, trading and wallet functionalities. Particularly, we have evaluated VetSC on 30 randomly selected widgets in 18 webpages from 13 top marketplace DApps at *dapp.com*, such as *LooksRare*, *OpenSea* and *X2Y2*, which cause the highest volumes of transactions in 24 hours (up to $176M). Our manual verification shows that we can correctly interpret 90% of the widgets. To our study, we can achieve this relatively high accuracy fundamentally because the interfaces of popular DApps have been designed in a user-friendly manner and thus can convey messages in a very clear way. For instance, widgets for placing bids often contain straightforward phrases such as "place a bid" or "make an offer". We do observe misclassifications, which are caused by vague widget text (e.g., "Create") plus overly broad contexts (e.g., "bid", "Highest", etc.).

**Discussion and Limitation.** NLP alone cannot easily extract textual semantics, but our UI analysis is viable because: (a) most widgets of well-intentioned, usable DApps bear straightforward and distinguishable meanings – the fact that a majority of our randomly selected widgets can be correctly interpreted has confirmed the expressiveness of widget text; (b) our UI text analysis is a guided process – we use domain knowledge, such as DOM-tree layout, mined function/contract-level keywords, as guidance to selectively obtain UI text and infer its semantics in a targeted way.

Nevertheless, UI analysis and text mining are known to be challenging in the areas of information retrieval and software engineering. To capture precise contextual information, researchers have attempted to use many heuristics such as website layout [43],

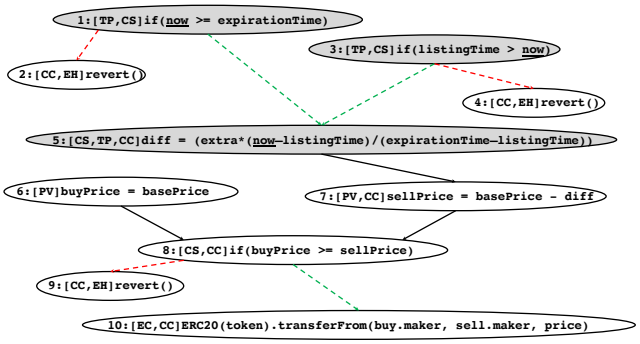


**Figure 7: BMG of Dutch auction in *OpenSea***

distance between widgets [32] or text quality [12] but have not presented any general solution; web pages and contents can be dynamically generated and thus cannot be observed unless certain JS code is triggered. In addition to the limitation of vanilla UI analysis, our custom method may also be limited by our design choices. First, while most of widget texts are short phrases, there indeed exist widgets with lengthy texts. Thus, it would be necessary (but non-trivial) to identify core semantic entities at a sentence level rather than directly applying word-level interpretation. Second, the exploitation of domain knowledge may fundamentally constrain our ability to explore a broader range of textual semantics. Furthermore, our UI analysis is based upon text interpretation and thus cannot handle cases where UI components do not contain any text but use icons to convey information. It may require more advanced techniques [61] to learn icon semantics and address this limitation.

## 5.5 Handling Large Apps

We evaluate VetSC with a top NFT marketplace app: *OpenSea* [8] to showcase the scalability. The analysis begins from the web interface of *OpenSea*, and can identify the textual semantics of "Place bid" to be "bidding in an auction", and correctly correlate this button to its backend Dutch auction function `atomicMatch()`. Next, we build the BMG (Figure 7) for this function starting from its transaction properties (e.g., *timestamp* and *buyer/seller*). Our graph captures the Dutch auction logic: it calculates the current selling *price* by subtracting a timing-related *diff* amount from the starting price, and eventually transfers funds from the *buy[er]* to the *sell[er]* using an *ERC20* API – `transferFrom(from, to, amount)`. In the meantime, it also enforces safety checks for expiration time and bid prices. The essential logic and safety checks follow our predefined rules for Dutch auctions in Table 7.

Our semantic inference works for large apps because the core logic of even complex apps may still be dense – the core auction logic in the *OpenSea* contains only ~20 LOC (out of 1836). Nevertheless, these 20 lines of code span multiple functions and are hidden in a large volume of less relevant code. It would be non-trivial for humans to precisely and completely discover even such succinct core logic. Admittedly, if core logic becomes overly complex, it may also affect the precision and efficiency of our model checking.

## 5.6 DeFi Apps & Impact of Cross-Contract Calls

To demonstrate the generalizability of our method for modeling contract semantics, we further explore the possibility of applying VetSC to decentralized finance (DeFi) apps.

**Effectiveness of Semantic Modeling.** Emerging DeFi apps enable intricate financial services on blockchains using intensive cross-contract transactions. In contrast, our evaluation shows that their core business logic can still be captured by our *business model graphs*. We have evaluated our method using a popular DeFi app *UniswapV2* [10]. Figure 8 illustrates the graph we compute from the *swap ether for token* logic in the *UniswapV2*.

Specifically, this swap logic can exchange a user's ether for specific tokens through possibly a series of intermediate token exchanges, and transfer the tokens to a user specified address. It is implemented in two essential functions across two separate contracts: `swapExactETHForTokens()` in the *Periphery* Contract and `swap()` in the *Core* Contract, and some other helper functions. The former calculates the token amounts, performs security checks, and makes a cross-contract call to the latter, which then interacts with domain-specific token contracts and enables the swap. To generate this graph, we start from two transaction properties `msg.value` (the amount of ether, V#2) and `msg.data` (containing the target address, V#8), as well as a global token exchange pool `factory` (V#1). The *Periphery* Contract receives the user's ether (V#2), transfers it to the *Uniswap* token (*WETH*) contract (V#4), and updates the exchange pool (V#3). It then computes the amounts of tokens it needs to pay the user (V#6), and eventually calls the `swap()` function in the *Core* Contract in order to transfer tokens to the user's account "`to`" (V#8). The *Core* validates the required token amounts (V#10,12,13), and then performs consecutive exchanges between pairs of tokens, and finally transfers the requested tokens to the specified address (V#14,15). While the two contracts contain 647 lines of code, the core logic in our graph involves only ~30 LOC.

The logic in the graph can be captured by our *Uniswap* spec in Table 7. Particularly, we specify that a token swap operation must implement four essential activities: (E1) a user sends ether to the exchange pool; (E2) the *Uniswap* contract transfers tokens to a destination Ethereum address; (E3) the amount of tokens depends on the amount of the given ether; (E4) the destination address is specified by the user. The constraints E1 and E2 are satisfied by V#4 and V#15, while E3 and E4 are fulfilled by the data dependencies between V#2 and V#15, and between V#8 and V#15, respectively. Our safety vetting verifies if certain checks for token amounts exist.

**Cross-Contract Calls.** Despite the fact that VetSC can capture DeFi logic in theory, it cannot handle DApp logic that spans multiple contracts – which is present in the aforementioned example and commonly observed in DeFi apps [60]. Essentially, this is because VetSC adopts the same Effectively External Callback Free (EECF) model used in VerX [45], and thus considers external calls as NOPs.

However, our evaluation on the top 600 GitHub DApp projects has shown that the impact of cross-contract calls is currently limited. Among the top projects we have collected from the six app categories, only a small portion (7.7%) has actually made calls to other contracts. Moreover, very few of essential business logics (3.9%) are implemented across multiple contracts using inter-contract calls. Indeed, some voting or trading contracts (e.g., `aragon-parliament`) authenticate users (voters, buyers) in one contract and finalize business logic (increase votes, close sales) in another. Nevertheless, such two-step logic is actually implemented easily using solely local calls in most of the apps, and thus can still be largely captured by VetSC.

**Table 7: Partial Specifications for Dutch Auction and UniswapV2**

| | Function | Spec Type | Formal Spec | Explanation |
|---|---|---|---|---|
| Dutch Auction | Bidding | Essential#1 | $\square(\neg current\_time \xrightarrow{data} \Diamond(\neg current\_sell\_price))$ | Current sell price is data-dependent on current time |
| | | Essential#2 | $\square(current\_bid >= current\_sell\_price \rightarrow \Diamond(auction.owner.send(current\_sell\_price)))$ | Send current sell price to the owner when a successful bid exists |
| | | Safety#1 | $\square(current\_time >= deadline \rightarrow \Diamond(execution\_state := revert))$ | No bidding on expired auction |
| | | Safety#2 | $\square(current\_bid < current\_sell\_price \rightarrow \Diamond(execution\_state := revert))$ | Current bid must be higher than the current sell price |
| UniSwapV2 | Swap Ether w/ token | Essential#1 | $\square pool.send(ether\_amount)$ | Send Ethers to the pool |
| | | Essential#2 | $\square transfer(pool, dest, token\_amount)$ | Transfer the tokens from the pool to the destination |
| | | Essential#3 | $\square(\neg ether\_amount \xrightarrow{data} \Diamond(\neg token\_amount))$ | The amount of token is data-dependent on the sent Ethers |
| | | Essential#4 | $\square(\neg calldata \xrightarrow{data} \Diamond(\neg dest))$ | The destination is designated by user |
| | | Safety#1 | $\square(token\_amount < 0 \rightarrow \Diamond(execution\_state := revert))$ | The amount of token needs to be non-negative |
| | | Safety#2 | $\square(token\_amount > reserve \rightarrow \Diamond(execution\_state := revert))$ | The amount of token needs to be smaller than what pool possesses |

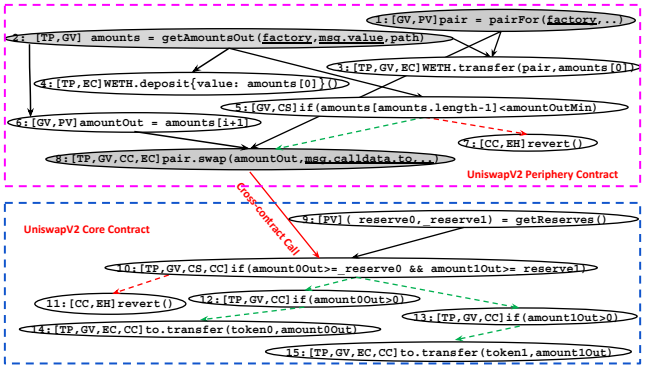


**Figure 8: BMG of swap logic in *UniswapV2***

## 5.7 Runtime Efficiency

Our model construction uses dynamic analysis to find smart contract calls in DApps and static analysis to build models. The dynamic analysis is constrained by the response time of DApps and on average takes 5 minutes to process an app; static analysis takes on average 280 seconds to build a model. Our model checking process is very time consuming due to its iterative nature, and its runtime is roughly proportional to the amount of specifications. In our current setting, VetSC takes on average 470 seconds to check one app.

# 6 RELATED WORK

**Verifying the Safety of Smart Contracts.** Efforts [23, 26, 33, 34, 39, 45, 50, 52, 53, 58, 59] have been made to automatically verify the safety of smart contract code. While early work aimed to discover syntax based low-level errors [23, 26, 34, 39, 50], more recent studies [33, 45, 53, 58] have started to investigate the semantic-level defects in smart contracts that can cause fairness issues. For instance, ZEUS [33] claimed that a fair "without-reserve" auction must not allow the *seller* to bid; VerX [45] stated that a crowdsale is safe only if *refunds* are offered to investors once the sale has failed.

**Recovering Semantics for Program Analyses.** Automated semantics recovery has been well studied in multiple domains including virtual machine introspection [18, 25, 31], memory forensics [13, 16, 19, 37] and low-level code interpretation [35, 40]. The contribution of our work lies in the fact that we adopt this technique in the new context of smart contracts. Thus, we need to invent a novel algorithm to distill contracts and transaction -specific properties that can reflect their semantics. In addition, we take a step further to recover business logic level semantics.

**Correlating Descriptive Text to Program Behaviors.** Efforts have also been made to correlate texts to sensitive API semantics in Android [29, 44, 48] and IoT apps [57]. WHYPER [44] used NLP technique to identify sentences that describe the need for a given permission. AutoCog [48] developed a learning-based algorithm to automatically derive a model that correlates textual descriptions with Android permissions; AsDroid [29] further inferred the semantics of the text on those widgets that are associated with the top level functions. SmartAuth [57] combined NLP and program analysis to distill the contextual semantics of IoT apps, and compared them with the textual semantics in source code and code annotations.

**Using GUI for Security Purposes.** Many techniques [46, 47, 49, 51] have been proposed to perform access control for device sensors via GUI events. Particularly, efforts have been made in the domain of mobile applications to identify sensitive user inputs from GUI based on descriptive texts [11, 28, 41] and icons [61]. UIRef [11] used adjacent five components in UI layout as context to infer semantics and address ambiguity. SUPOR [28], UIPicker [41] and IconIntent [61] considered context to be texts on the single view hosting the target widget. In contrast to traditional UI-based apps, the frontend UI and backend code of DApps are only loosely connected, and thus require a holistic approach to analyze their correlations.

# 7 CONCLUSION

We propose VetSC, a novel UI-driven, program analysis guided model checking technique that can automatically extract contract semantics in decentralized apps so as to enable targeted safety vetting. We build *business model graphs* to model core business logic behaviors; we retrieve textual semantics from DApp UI to assist in automatically customizing safety specifications. VetSC has been applied to 34 real-world DApps and has discovered 19 novel safety issues.

# ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their constructive comments. We would also like to thank Yihua Xu, William Bodell, Sajad Meisami, and Taiji Li, for their help during the evaluation data collection process. This work was supported in part by NSF OAC-2115167, NSF DGE-2041960, DARPA HR00112120009 Cooperative Agreement, NSFC-61872180, and a USHE Deep Technology Initiative Grant. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

# REFERENCES

[1] 2020. King of the Ether Throne. https://github.com/kieranelby/KingOfTheEtherThrone/. (2020).

[2] 2020. MetaMask: Brings Ethereum to your browser. https://metamask.io/. (2020).
[3] 2020. Octopus: Security Analysis tool for Blockchain Smart Contracts. https://github.com/quoscient/octopus/. (2020).
[4] 2020. Stanford Log-linear Part-Of-Speech Tagger. https://nlp.stanford.edu/software/tagger.shtml. (2020).
[5] 2021. Ethereum Virtual Machine Opcodes. https://ethervm.io/. (2021).
[6] 2021. NuSMV: a new symbolic model checker. http://nusmv.fbk.eu/. (2021).
[7] 2022. Block and Transaction Properties. https://docs.soliditylang.org/en/v0.8.11/units-and-global-variables.html#block-and-transaction-properties/. (2022).
[8] 2022. Discover, collect, and sell extraordinary NFTs. https://opensea.io/. (2022).
[9] 2022. selenium. https://www.selenium.dev/. (2022).
[10] 2022. Uniswap Docs. https://docs.uniswap.org/protocol/V2/concepts/protocol-overview/smart-contracts/. (2022).
[11] Benjamin Andow, Akhil Acharya, Dengfeng Li, William Enck, Kapil Singh, and Tao Xie. 2017. UiRef: Analysis of Sensitive User Inputs in Android Applications. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 23–34.
[12] Michael Bendersky, W Bruce Croft, and Yanlei Diao. 2011. Quality-Biased Ranking of Web Documents. In *Proceedings of the fourth ACM international conference on Web search and Data Mining*. 95–104.
[13] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. 2009. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*.
[14] ChainSecurity. 2021. VerX API. (2021). http://verx.ch/docs/api.html
[15] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-Aware Malware Detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 32–46.
[16] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. 2012. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *Proceedings of the 21st USENIX Conference on Security Symposium (USENIX Security'12)*.
[17] Michael del Castillo. 2016. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft. (2016).
[18] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *2011 IEEE Symposium on Security and Privacy*.
[19] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust Signatures for Kernel Data Structures. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*.
[20] Ebay. 2022. Ebay: Ending a listing. https://www.ebay.com/help/selling/listings/creating-managing-listings/cancelling-listing?id=4146/. (2022).
[21] Ethereum. 2021. INTRODUCTION TO DAPPS. https://ethereum.org/en/developers/docs/dapps/. (2021).
[22] Ittay Eyal and Emin Gün Sirer. 2014. Majority is Not Enough: Bitcoin Mining is Vulnerable. In *International Conference on Financial Cryptography and Data Security*. Springer, 436–454.
[23] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. {ETHBMC}: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*.
[24] Jake Frankenfield. 2019. 51% Attack. https://www.investopedia.com/terms/1/51-attack.asp. (2019).
[25] Tal Garfinkel, Mendel Rosenblum, et al. 2003. A Virtual Machine Introspection based Architecture for Intrusion Detection.. In *NDSS*. 191–206.
[26] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. In *Procceedings of The 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2018)*.
[27] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. 2015. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *24th USENIX Security Symposium (USENIX Security 15)*. 129–144.
[28] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. {SUPOR}: Precise and Scalable Sensitive User Input Detection for Android Apps. In *24th USENIX Security Symposium (USENIX Security 15)*. 977–992.
[29] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction *(ICSE 2014)*.
[30] Christopher D. Manning Jeffrey Pennington, Richard Socher. 2020. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/. (2020).
[31] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*.
[32] Hongyan Jing and Evelyne Tzoukermann. 1999. Information Retrieval Based on Context Distance and Morphology. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and Development in Information Retrieval*. 90–96.
[33] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 2018 Network and Distributed System Security Symposium*.
[34] Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security'18)*.
[35] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs.. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.
[36] Zhiqiang Lin, Junghwan Rhee, Chao Wu, Xiangyu Zhang, and Dongyan Xu. 2012. Dimsum: Discovering Semantic Data of Interest from Un-mappable Memory with Confidence. In *Proceedings of NDSS Symposium 2012*.
[37] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*.
[38] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards Automated Verification of Smart Contract Fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 666–677.
[39] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*.
[40] Stephen McLaughlin and Patrick McDaniel. 2012. SABOT: Specification-based Payload Generation for Programmable Logic Controllers. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*.
[41] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. Uipicker: User-input Privacy Identification in Mobile Applications. In *24th USENIX Security Symposium (USENIX Security 15)*. 993–1008.
[42] State of the DApps. 2022. DApp Statistics. https://www.stateofthedapps.com/stats. (2022).
[43] Harrie Oosterhuis and Maarten de Rijke. 2018. Ranking for Relevance and Display Preferences in Complex Presentation Layouts. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR '18)*.
[44] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Conference on Security (USENIX Security 13)*.
[45] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (Oakland)*.
[46] Giuseppe Petracca, Ahmad-Atamli Reineh, Yuqiong Sun, Jens Grossklags, and Trent Jaeger. 2017. Aware: Preventing Abuse of Privacy-Sensitive Sensors via Operation Bindings. In *26th USENIX Security Symposium (USENIX Security 17)*. 379–396.
[47] Giuseppe Petracca, Yuqiong Sun, Ahmad-Atamli Reineh, Patrick McDaniel, Jens Grossklags, and Trent Jaeger. 2019. Entrust: Regulating Sensor Access by Cooperating Programs via Delegation Graphs. In *28th USENIX Security Symposium (USENIX Security 19)*. 567–584.
[48] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. AutoCog: Measuring the Description-to-Permission Fidelity in Android Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*.
[49] Talia Ringer, Dan Grossman, and Franziska Roesner. 2016. Audacious: User-driven Access Control With Unmodified Operating Systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 204–216.
[50] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *Proceedings of the 2019 Network and Distributed System Security Symposium*.
[51] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J Wang, and Crispin Cowan. 2012. User-driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 224–238.
[52] Evgeniy Shishkin. 2019. Debugging Smart Contract's Business Logic Using Symbolic Model Checking. *Programming and Computer Software* 45, 8 (2019), 590–599.
[53] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (Oakland)*.
[54] Solidity. 2020. Solidity. https://solidity.readthedocs.io/en/v0.6.1/. (2020).
[55] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps.. In *NDSS*.
[56] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *IEEE S&P*.

[57] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. 2017. Smartauth: User-Centered Authorization for the Internet of Things. In *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security 17)*.

[58] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*.

[59] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. 2018. Formal Specification and Verification of Smart Contracts for Azure Blockchain. *arXiv preprint arXiv:1812.08829* (2018).

[60] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. 2021. DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications. *arXiv preprint arXiv:2104.15068* (2021).

[61] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. Iconintent: Automatic Identification of Sensitive UI Widgets Based on Icon Classification for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 257–268.

[62] Zethyr. 2021. Zethyr Exchange. https://www.dapp.com/app/zethyr-exchange/. (2021).