Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications

Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li,
Fengyuan Xu, Mu Zhuang

CCS’ 22

Background

- Smart contract

```
pragma solidity >=0.7.0 <0.9.0;

contract Storage {

    uint256 number;

    function store(uint256 num) public {
        number = num;
    }

    function retrieve() public view returns (uint256){
        return number;
    }
}
```

Source code

608060405234801561001057600080fd5b506101
50806100206000396000f3fe608060405234801561
001057600080fd5b50600436106100365760003
560e01c80632e64cec11461003b5780636057361
d14610059575b600080fd5b610043610075565b6
040516100509190610d9565b60405180910390f
35b61007360048036038101906100 6e919061009
d565b61007e565b005b60008054905090565b80
600081905550505b600081359050610097816 1
0103565b92915050565b600060208284031215 6
100b3576100b26100fe565b5b6……

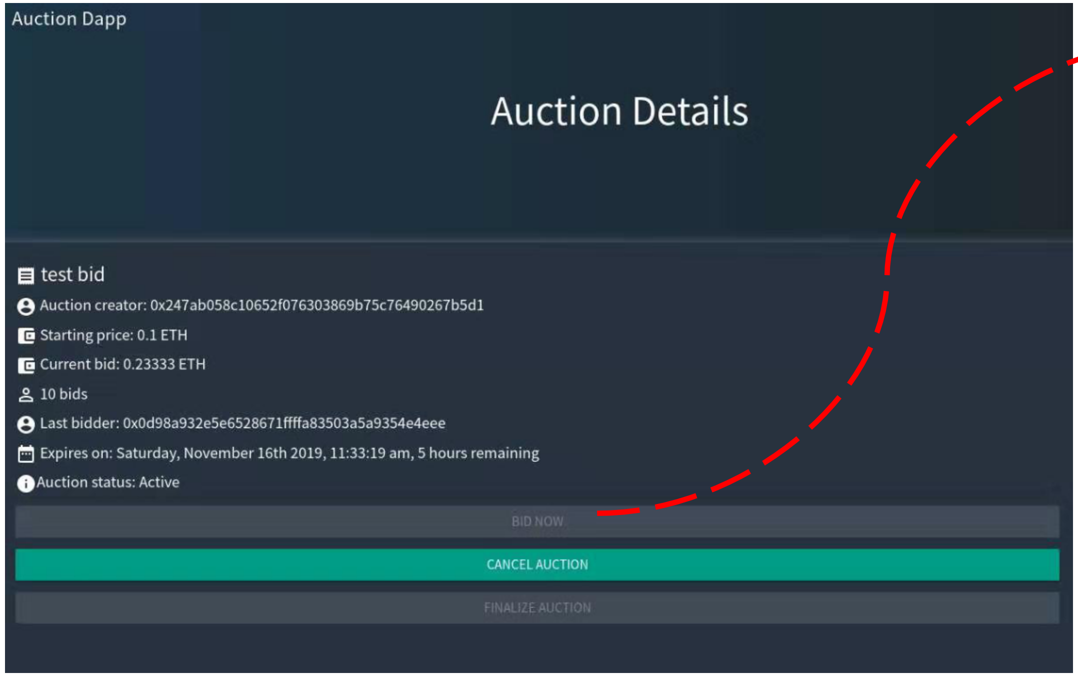
Bytecode

- Dapp

front-end UI

smart contract

Business Logic Correctness



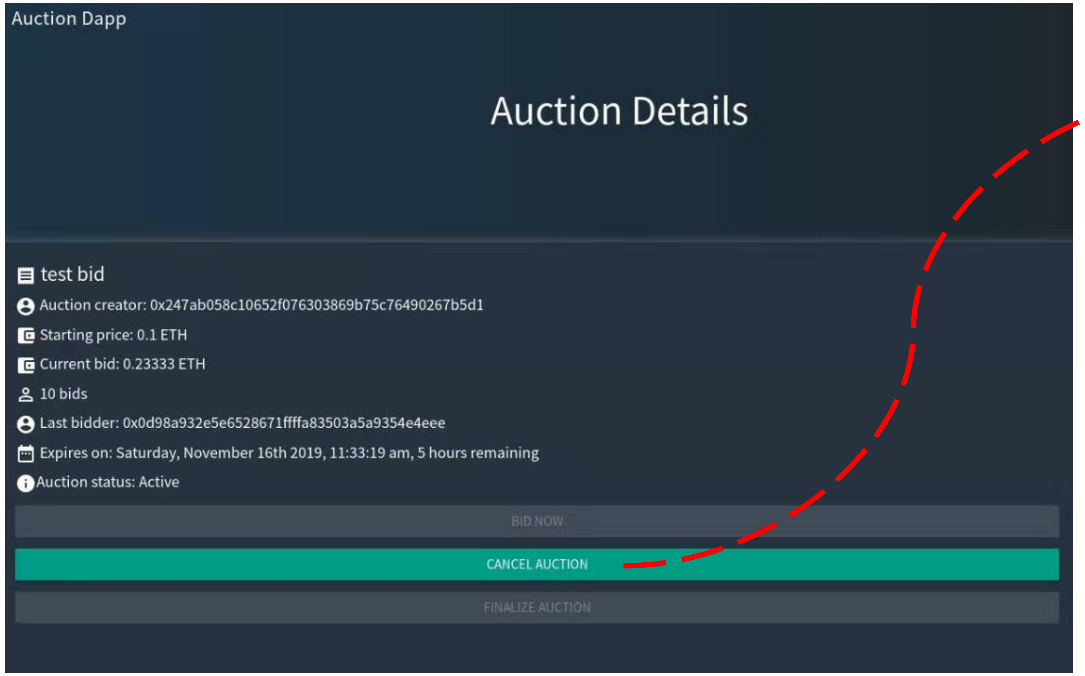

```
function bidOnAuction(uint _id) public payable {
    uint256 ethAmountSent = msg.value;

    // owner cannot bid on his/her own merchandise
    Auction memory myAuction = auctions[_id];
    if(myAuction.owner == msg.sender) revert();

    // check whether auction has expired
    if(block.timestamp > myAuction.deadline) revert();

    // check whether previous bids exist
    uint bidsLength = accepted[_id].length;
    uint256 tempAmount = myAuction.startPrice;
    Bid memory lastBid;
    if(bidsLength > 0) {
        lastBid = accepted[_id][bidsLength-1];
        tempAmount = lastBid.amount;
    }

    // check if bid price is greater than the current highest
    if(ethAmountSent < tempAmount) revert();

    // add the new bid to auction state
    Bid memory newBid;
    newBid.from = msg.sender;
    newBid.amount = ethAmountSent;
    accepted[_id].push(newBid);
    emit BidSuccess(msg.sender, _id);
}
```

owner cannot bid

auction has not expired

Business Logic Correctness

only the merchandise owner can cancel the auction




```
function cancelAuction(uint _id) public isOwner(_id) {
    Auction memory myAuction = auctions[_id];
    uint bidsLen = accepted[_id].length;

    // refund the last bid, if prior bids exist
    if(bidsLen > 0) {
        Bid memory lastBid = accepted[_id][bidsLen - 1];
        if(!lastBid.from.send(lastBid.amount)) revert();
    }
    auctions[_id].active = false;
    emit AuctionCanceled(msg.sender, _id);
}
```

Safety: only the owner can cancel her auction ✓

Fairness: no valid bid has been received. ✗

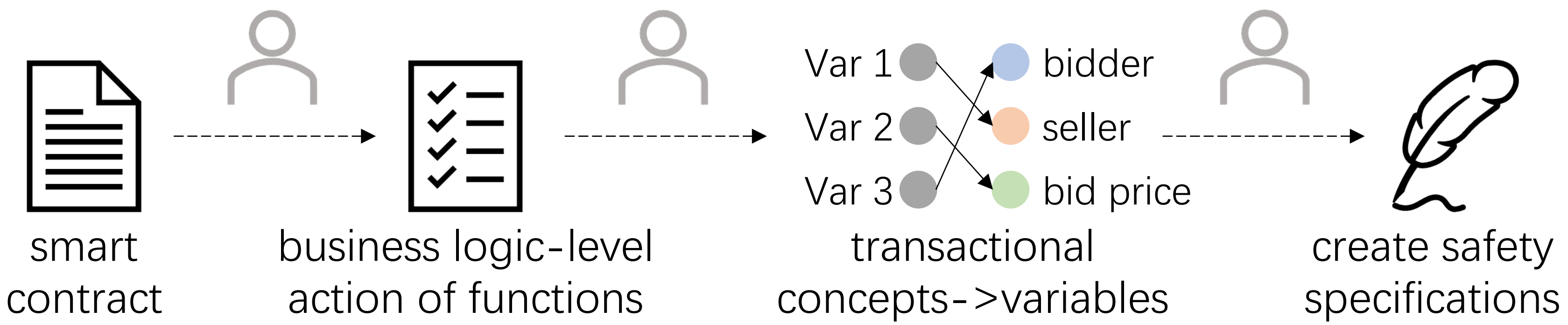
Two types of business logic are incorrect：

1. Incorrect logic implementation
2. UI-logic inconsistency

Problem& Challenge

Problem:

understand the semantics of contracts and create contract-specific safety specifications, and apply high-level safety specs to corresponding low-level smart contract code.



Disadvantages：

- Missing source code
- Duplication of effort
- Easy to omit

Challenge:

- Automatic advanced semantic recovery
- Automatic selection of security vetting specifications

Approach

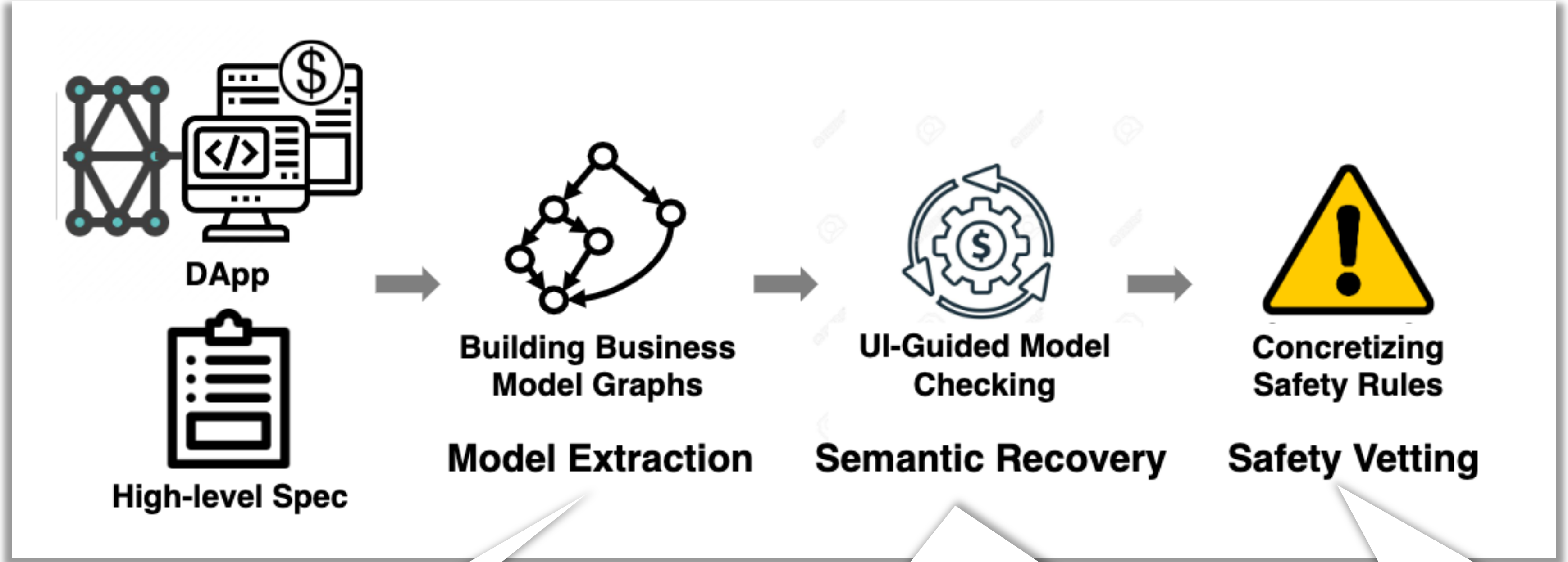
- Automatic advanced semantic recovery

perform static program analysis to extract business model graphs from contract functions that can capture intrinsic business logic across critical transaction properties， the method also excludes irrelevant code and reduce search space for model checkers

- Automatic selection of security vetting specifications

leverage textual information collected from DApp user interfaces, describing high-level underlying contract logic behaviors

Overview



extract the business model of every contract function

1. correlate each contract function to a front-end widget.
2. NLP infer the business logic the widget describes.
3. Selective consistency check

If consistent, check target functions against custom safety policies

Model Extraction

Step 1: formally define semantics model - Business Model Graph (BMG)

➢ Key factors:

Transaction Property; Global Variables; Dataflow; Condition Check; Cryptocurrency/Token Transfer

➢ BMG is a directed graph $G = (V, E, \alpha, \beta)$ over a set of statements Σ and a set of relations R

- set of vertices V ➡ the statements in Σ
- set of edges E ➡ causal dependencies between statements
- node labeling function α ➡ node ↔ its label

ID	attributes	a smart contract statement

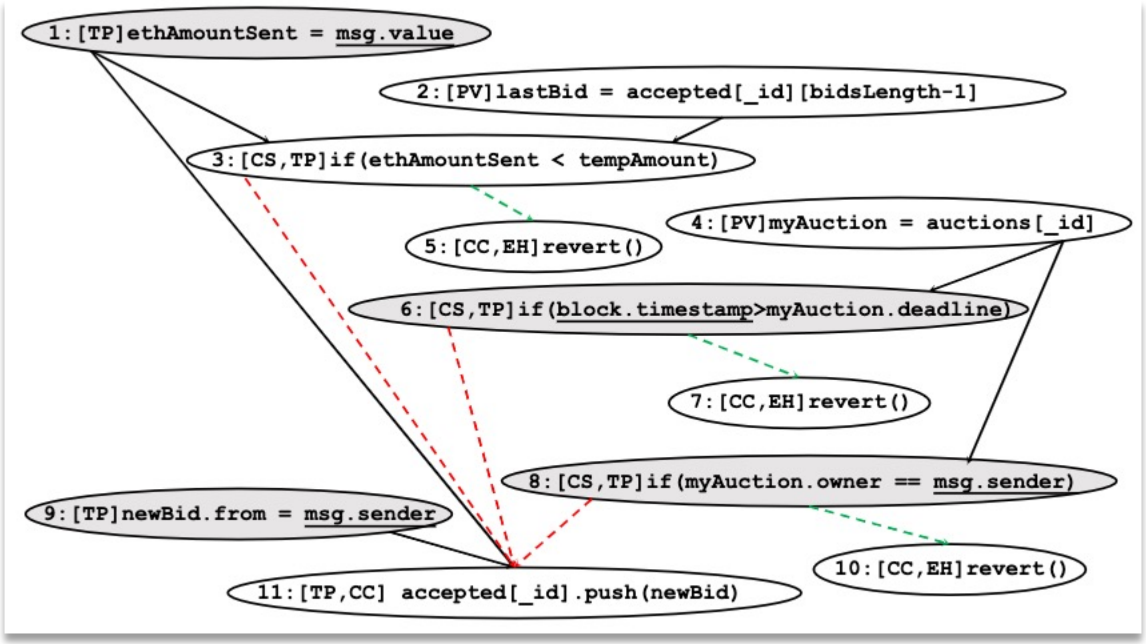
1. transaction property [TP]
2. global variable [GV]
3. conditional statement [CS]
4. variables in predicates [PV]
5. conditional clause [CC]
6. conditional clause [CC]
7. call parameter [CP]
8. exception handling [EH]

- edge labeling function β ➡ edge ↔ its label

⟶ data dependency ⇢ true branch ⇢ false branch

Model Extraction

```
function bidOnAuction(uint _id) public payable {
    uint256 ethAmountSent = msg.value;

    // owner cannot bid on his/her own merchandise
    Auction memory myAuction = auctions[_id];
    if(myAuction.owner == msg.sender) revert();

    // check whether auction has expired
    if(block.timestamp > myAuction.deadline) revert();

    // check whether previous bids exist
    uint bidsLength = accepted[_id].length;
    uint256 tempAmount = myAuction.startPrice;
    Bid memory lastBid;
    if(bidsLength > 0) {
        lastBid = accepted[_id][bidsLength-1];
        tempAmount = lastBid.amount;
    }

    // check if bid price is greater than the current highest
    if(ethAmountSent < tempAmount) revert();

    // add the new bid to auction state
    Bid memory newBid;
    newBid.from = msg.sender;
    newBid.amount = ethAmountSent;
    accepted[_id].push(newBid);
    emit BidSuccess(msg.sender, _id);
}
```



arrows point to the destinations

data dependency   true branch   false branch

Model Extraction

Step 2: Model Constuction

Algorithm 1 Graph Construction

```
procedure BuildBMG(SC)
    BMG ← ∅
    PointsToAnalysisForGlobalVariables(SC)
    INIT ← InitialEssentialStmt(SC)
    for ∀stmt ∈ INIT do
        SINK ← ForwardDataFlowAnalysis(stmt)
        for ∀sink ∈ SINK do
            BMG ← BMG ∪ < stmt, sink >
            SVAR ← ∅
            if sink is ConditionCheck then
                SVAR ← GetPredVars(sink)
                BRANCH ← ControlDepAnalysis(sink)
                OP ← FindStorageOpsOrExceptions(BRANCH)
                for (op, cond) ∈ OP do
                    BMG ← BMG ∪ < sink, op >_cond
                end for
            else if sink is Transfer then
                SVAR ← GetAddr(sink) ∪ GetValue(sink)
            else if sink is GlobalVariableWrite then
                SVAR ← GetOffset(sink) ∪ GetValue(sink)
            end if
            for ∀svar ∈ SVAR do
                src ← BackwardDataflowAnalysis(svar)
                BMG ← BMG ∪ < src, svar >
            end for
        end for
    end for
    return BMG
end procedure
```

control dependency

condition branch
True or False?

data dependency

analysis on memory operations that access global variables

collects stmt with transaction properties or global variables

find sink, and add <stmt, sink>

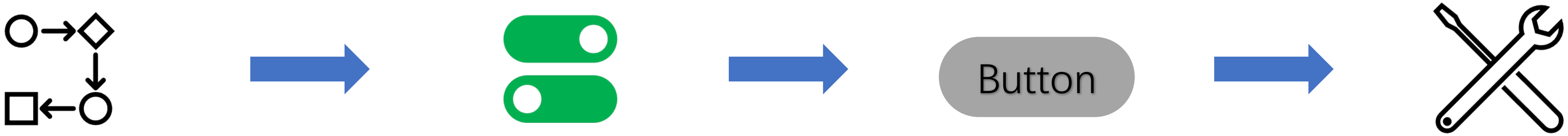
locate var in sink, backward analysis, and add <src, svar>

Semantic Recovery

Method: collect UI information relevant to each function, and then use model checking to match the business logic inferred from the user interface with the BMG of the corresponding function.

Step 1: UI Semantic Inference

➢ Identifying Contract Function related UI Text



fuzzing to iteratively explore each widget

monitor the function it invokes

collect texts from these widgets

contextual collection through existing tools

➢ Inferring UI Semantics via NLP

- build semantic templates for Dapps: contract-level and function-level
- matching UI Text to templates using NLP

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*

Semantic Recovery

Step 2: Model Checking & Semantic Recovery

develop specifications for various types of Dapp and check whether the business logic of the Dapp corresponding to the UI matches the contract code

- Specification
  - Essential Logic, which specifies the basic – yet unique and intrinsic
  - Safety Rule that further specifies the safety constraints for particular business logics

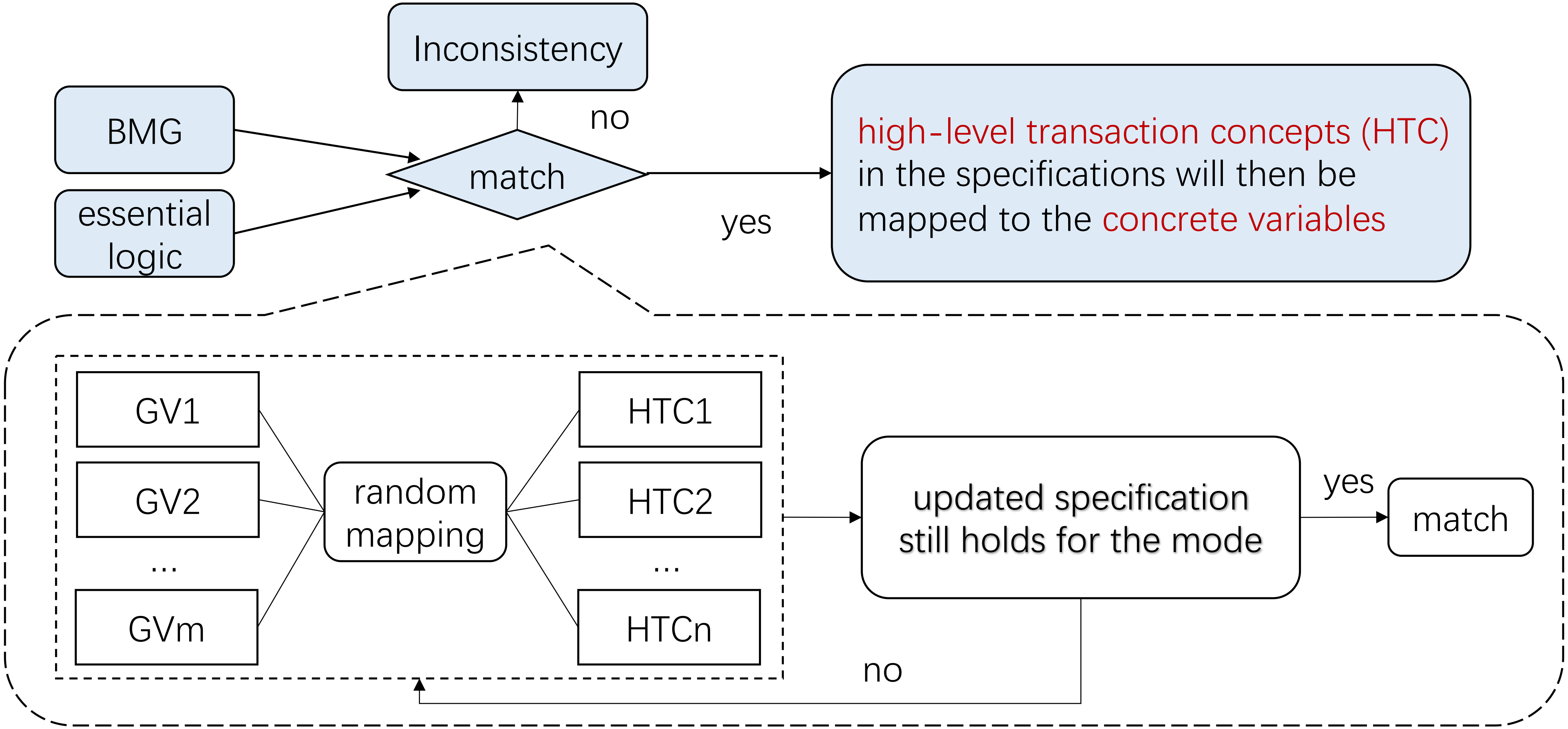
high-level transaction concepts

	Function	Spec Type	Formal Spec	Explanation
Auction	Bidding	Essential#1	$\Box(current\_bid > highest\_bid \rightarrow \Diamond(highest\_bid := current\_bid \land highest\_bidder := current\_bidder))$	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	$\Box(requester \mathrel{!=} auction.owner \rightarrow \Diamond(execution\_state := revert))$	Only owner can cancel an auction
		Safety#2	$\Box(highest\_bidder \mathrel{!=} null \rightarrow \Diamond(execution\_state := revert))$	Cannot cancel when a valid bid exist

Semantic Recovery

Step 2: Model Checking & Semantic Recovery

➢ A Model Checking Problem & Recovering Semantic



Semantic Recovery

Step 2: Model Checking & Semantic Recovery

➢ A Model Checking Problem & Recovering Semantic

	Function	Spec Type	Formal Spec	Explanation
Auction	Bidding	Essential#1	$\Box(current\_bid > highest\_bid \rightarrow \Diamond(highest\_bid := current\_bid \land highest\_bidder := current\_bidder))$	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

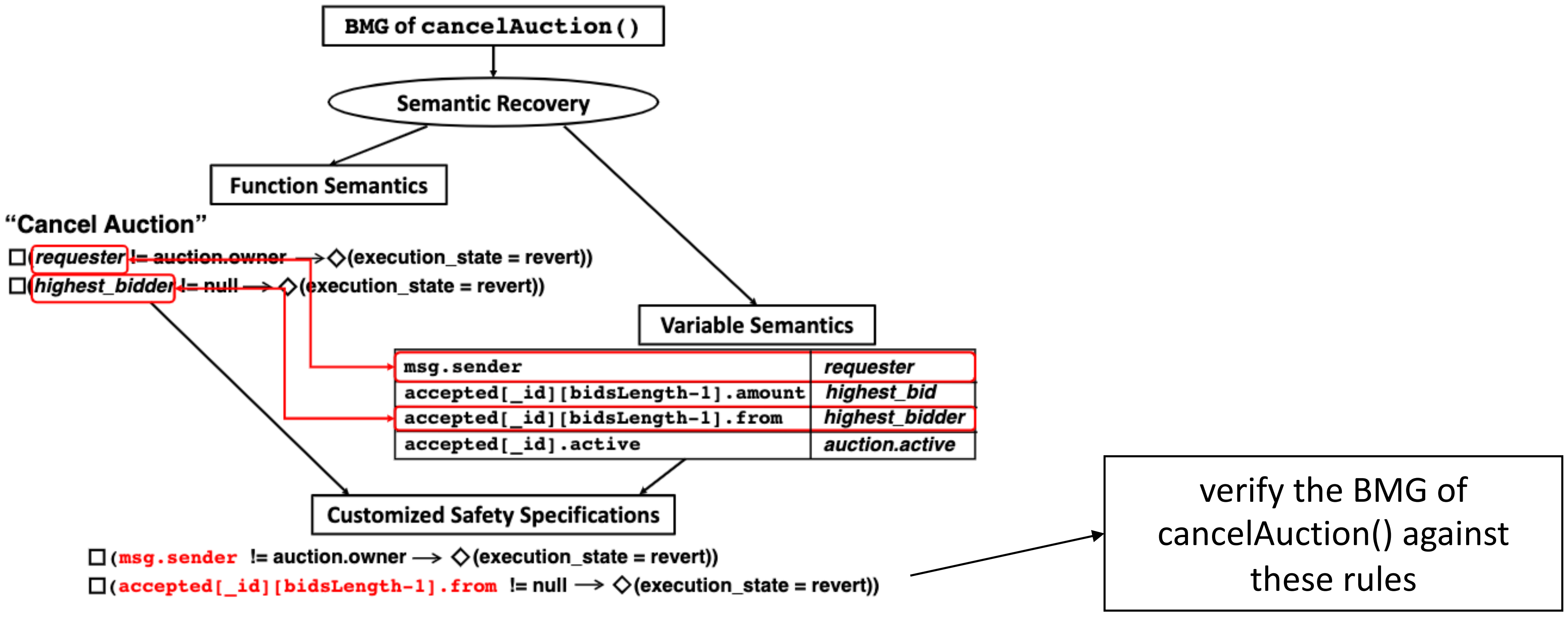
Two-level semantics

- Function Semantics
- Variable Semantics

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*

Safety Vetting

- function semantics: informs us which set of safety policies need to be applied to a specific function
- variable semantics: tells us how to customize them



#	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/Absence"*	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

Evaluation

- Effectiveness of UI Semantic Inference
  - UI Analysis on Random Widgets (185 widgets)
    - ✓ 150 widgets can trigger smart contract functions → Correct Rate: 82.5%
    - ✓ 34 widgets that do not lead to contract calls → Error rate: 65%
  - UI Analysis on Large Apps
    - ✓ 13 top marketplace Dapps, 30 randomly selected widgets → Correct Rate: 90%

- Runtime Efficiency
  - dynamic analysis to find smart contract calls: 5 minutes
  - static analysis to build models: 280 seconds

Conclusion

Problem	Business Logic Correctness	
Approach	build BMG to model core business logic behaviors	retrieve textual semantics from UI to assist customizing safety specifications
Effect	Enable targeted safety vetting	