Reinforcement Learning-Guided Symbolic Execution for Efficient and Exploitable Smart Contract Analysis

Zhaoxuan Li[1,2,3] Ziming Zhao[4] Siqi Lu[5] Rui Zhang[1*] Wenhao Li[6] Xiaofei Yue[7] Fan Zhang[4]

[1]State Key Laboratory of Cyberspace Security Defense, IIE, CAS, Beijing, China
[2]Guizhou Key Laboratory of NewGen Cyberspace Security, Guiyang, China
[3]Laboratory for Advanced Computing and Intelligence Engineering, Wuxi, China
[4]Zhejiang University, Hangzhou, China [5]Information Engineering University, Zhengzhou, China
[6]China Telecom Corp Ltd, Guangdong Research Institute, Guangzhou, China
[7]Beijing Institute of Technology, Beijing, China [*]Corresponding author
{lizhaoxuan, zhangrui}@iie.ac.cn

Abstract

Currently, frequent security incidents of Ethereum contracts have caused billions of dollars in losses. There is a pressing need to identify defective contract code and generate exploit call sequences to reproduce attacks and ensure detection accuracy. Nevertheless, the state-of-the-art (SOTA) detection methods based on symbolic execution and fuzzing cannot achieve the desired performance due to the state explosion problem caused by contract characteristics such as cross-contract calls and loop branches, especially in large wild contracts. To tackle this problem, we propose RSymX, an assembly strategy-guided symbolic execution for contracts that leverages reinforcement learning to dynamically balance code coverage and vulnerability discovery. Extensive experiments on open-sourced datasets demonstrate that RSymX achieves an overlap score of 92.39% and identifies many vulnerable wild instances that SOTAs misreport. Especially, its dynamic strategies improve the efficiency of state exploration by 2x~4x, and in some cases up to 10x, offering guidance for future symbolic-execution-based analyzers. The code and data are available in `https://github.com/Secbrain/RSymX`.

1 Introduction

Smart contracts, as the main component of blockchain systems, manage assets worth over $10 billion [38, 49]. The significant economic value incentivizes malicious users to seek and exploit contract vulnerabilities [41]. For instance, on May 15, 2024, the Compound fork project *Sonne Finance* suffered a loss of $20 million due to contract vulnerabilities such as *integer-overflow*. Therefore, effective checkers are crucial for ensuring the security (bug-free and well-designed) of contracts. However, the existing contract analysis methods based on various technologies, such as pattern matching (*e.g.,* Zeus [38], Slither [21], and SmartCheck [67]), and ML (*e.g.,* S-gram [45], DR-GCN [84], TMP [84], ContractWard [73], and VulHunter [42]) are challenging to generate exploit transaction sequences for reproducing the attacks and ensuring the reliability of results, when they identify vulnerable code snippets based on either contract source code or bytecode/opcode.

To this end, symbolic execution (*e.g.,* Oyente [49] and Mythril [63], and Manticore [51]) and fuzzing (*e.g.,* ILF [31] and Contractfuzzer [36]) based arts are designed not only to detect bugs, but to generate exploit transaction sequences that prove the flaws [22]. Nevertheless, as evaluated by SmartBugs [15, 23, 60] and Park [83], these tools have not delivered the claimed effectiveness due to the path explosion problem, which cased by the following contract characteristics. (i) The loop operations

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

and increasing functions. Owing to the ability to encode complex business logic, contracts are becoming increasingly complex in terms of logical code lines, state-changing public functions, and loop branches [58], so that they have up to tens of thousands paths, as depicted in Figure 1. (ii) Cross-contract calls and fallback functions. Frequent interactions between contracts require further expansion of the paths for called functions during the exploration process. (iii) Multiple transaction cooperation. Multiple consecutive transactions are required to explore contracts as comprehensively as possible, so that the number of state paths is growing exponentially. (iv) Additional overhead for vulnerability detection. The execution status of paths and the intermediate values of the vulnerability detection are required to be stored, so that the longer paths and more vulnerability detectors may out of the time limits and get failed results [58].

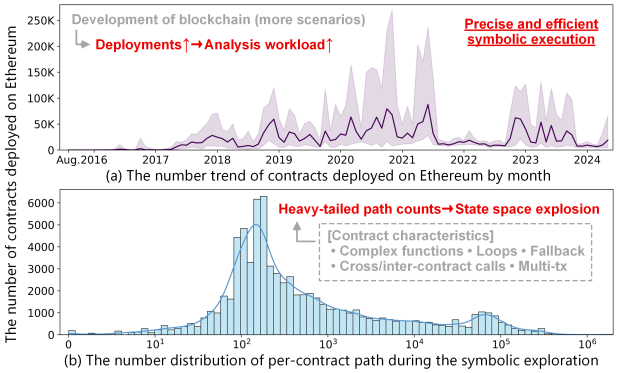


Figure 1: The number/path distribution of wild contracts.

Regarding to the larger analysis overheads, unguided search strategies (path pruning [63, 53] and fork parallel [83]) were used to improve the execution efficiency, yet massive states still cannot be fully handled. Instead, the coverage-guided (Learch [32] and ILF [31]) and vulnerability-guided (SmarTest [62], MythrilQL [72], TeEther [39]) methods filtered the suspicious paths based on the language models. Nevertheless, the code coverage and vulnerability discovery should be further balanced to get rid of local optima and achieve better guild effects [66]. In total, increasingly complex contracts make the current methods difficult to analyze as the state explosion problem and the aforementioned problems.

Contribution. To address the above problem, we propose RSymX, a dynamic strategy-guided symbolic execution framework. It is developed on the basis of SymX [43], which takes the contract source code, bytecode, or opcode as input, and eventually output defective code with vulnerabilities shown in Table 1, as well as the corresponding exploit call sequences. More importantly, RSymX automatically learns dynamic strategies that account for code coverage and vulnerability discovery through the designed hybrid reward mechanism and 3DQN (Delayed Double Deep Q Network) model to guide state exploration and discover vulnerabilities more effectively and quickly, thus mitigating the state space explosion in SymX. Compared with SOTA methods on several open-sourced datasets, it can identify contract vulnerabilities more accurately and comprehensively than SOTAs. Especially, the dynamic strategies accelerate the state exploration by 2x~4x (even up to 10x). Ablations indicate the principled guidance of our unique designs.

2 Motivation Examples and Problem Definition

2.1 Vulnerabilities in Smart Contracts

In order to characterize contract flaws in a unified manner, we adopt the vulnerability assessment scheme used in prior studies [41, 42]. It classifies findings into five levels: *High*, *Medium*, *Low*, *Informational* (*Info*), and *Optimization* (*Opt*).[1] Each finding is assessed from two perspectives: *risk degree* and *utilization difficulty*. The former measures the potential impact of a vulnerability on blockchain systems, users, and assets, while the latter reflects how difficult it is to exploit in practice. Following prior work, utilization difficulty is categorized as *exactly*, *probably*, and *possibly*, according to attack cost, triggering conditions, and exploitation complexity.

Table 1: The severity of vulnerabilities supported by RSymX.

ID	Vulnerability Name	Severity	ID	Vulnerability Name	Severity
RE	reentrancy-eth	High	UCL	unchecked-lowlevel	Medium
SU	suicidal	High	UCS	unchecked-send	Medium
CDC	controlled-delegatecall	High	TO	tx-origin	Medium
AS	arbitrary-send	High	TS	timestamp	Low
UIS	uninitialized-state	High	BP	block-other-parameters	Low
UIO	uninitialized-storage	High	LLC	low-level-calls	Info
TOD	tod-ether/receiver	High	MEZ	msgvalue-equals-zero	Info
IE	incorrect-equality	Medium	ST	send-transfer	Opt
IO	integer-overflow	Medium	BE	boolean-equal	Opt

[1]The assessment is detailed on `https://github.com/Secbrain/RSymX/tree/main/Severity_assessment`.

Table 2: Landscape for accelerating fuzzy testing and symbolic execution in terms of model, features, rewards, and objectives. Among them, 'Feat. I/E/S"' means Instruction/Environment/Sequence features. "Sem. L/X/F/Tx" refers to Loop/Cross-contract/fallback/multi-transaction semantics, where ½✓ indicates partial support.

Work	Model	Feat. I/E/S	Bug Reward	Coverage Reward	Objective	Action@Objects	Sem. L/X/F/Tx
Acceleration on fuzzy testing for contracts							
ILF [31]	MDP (GRU)	✓/✓/×	×	✓(new coverage)	Tx generation	Mutation@Txs	– (No symbolic modeling)
RLF [66]	DRQN (DNN)	✓/✓/×	✓ (findings)	✓(block coverage)	Tx generation	Mutation@Txs	
SMARTIAN [13]	Data-flow match	×/✓/×	×	✓(flow coverage)	Tx generation	Mutation@Txs	
Acceleration on symbolic execution for traditional programs (executor of KLEE [7])							
Q-KLEE [77]	Q-learning	×/×/×	×	✓ (new coverage)	Path schedule	Selection@States	×/×/×/×
Learch [32]	Learning (MLP)	✓/✓/×	×	✓ (leaf new coverage)	Path schedule	Selection@States	×/×/×/×
Acceleration on symbolic execution for contracts							
Maian [53]	Path pruning	–/–/–	×	✓ (new branch)	Path prune	Filter@Branches	×/×/×/½✓
Manticore [51]	Fork parallel	–/–/–	–	–	Path Parallel	Parallel@States	½✓/✓/✓/½✓
Park [83]	Fork parallel	–/–/–	–	–	Path Parallel	Parallel@States	½✓/×/×/½✓
Mythril [63]	Path pruning	–/–/–	×	✓ (new branch)	Path prune	Filter@Branch	½✓/×/×/½✓
TeEther [39]	A* (Critical paths)	✓/×/×	✓ (sensitive opcodes)	×	Path selection	Filter@Branches	×/×/×/½✓
SmarTest [62]	Learning (n-gram)	✓/×/×	✓ (sensitive opcodes)	×	Path selection	Filter@States	½✓/×/×/½✓
MythrilQL [72]	+Q-learning	×/×/×	✓ (sensitive opcodes)	–	Path schedule	Selection@States	½✓/×/×/½✓
SymX [43]	Fork parallel	–/–/–	–	–	Path Parallel	Parallel@States	✓/✓/✓/✓
RSymX	**+3DQN (MLP,...) +online updates**	✓/✓/✓	**✓ (severity-weighted findings)**	**✓ (time-normalized new coverage)**	**Path schedule**	**Selection@States**	✓/✓/✓/✓

2.2 Motivating Example and Problem Definition

To illustrate vulnerability detection at the source-code and bytecode/opcode levels, we use the *Crowdsale* contract in Listing 1 and discuss representative vulnerabilities (*c.f.,* Table 1) in terms of root cause and severity.[2]

Although users interact with *Crowdsale* via functions such as withdraw(), its flaws can be exploited under specific transaction sequences. (i) SU and AS. An attacker can call init() to become owner, and then either invoke destruct(Booloracle.address, 6) to reach selfdestruct() through Booloracle.allow(dvalue), or call setStatus(1) and withdraw() to transfer funds via attacker-reachable send operations. (ii) IO, UCS, UCL, and UIS. The updates of variables raised/deposits[msg.sender] may overflow. Unchecked send/low-level-call results may hide failed transfers. The uninitialized raised may corrupt accounting.

```
1  contract Booloracle { //Deployed with Crowdsale.
2    function allow(uint value) public returns (bool) {
3      if(value > 5){ return true; }
4      return false;
5    }
6    function() { ... } //This may be malicious.
7  }
8  contract Crowdsale {
9    address owner; bool tBlock = false; //Prevent RE.
10   uint256 raised, goal, status; //UIS.
11   mapping (address => uint256) deposits;
12   constructor(uint256 goalFund) public payable {
13     owner = msg.sender, goal = goalFund, status = 0;
14   }
15   function init() public {
16     owner = msg.sender; //Anyone can invoke it.
17   }
18   function setStatus(uint256 newStatus) public {
19     require((newStatus == 1 && raised >= goal) ||
         (newStatus == 2 && raised < goal));
20     status = newStatus;
21   }
22   function invest() public payable {
23     require(status == 0 && raised < goal);
24     deposits[msg.sender] += msg.value; //IO
25     raised += msg.value; //IO
26   }
27   function withdraw() public {
28     require(status == 1);
29     owner.send(raised); //AS, UCS and ST.
30   }
31   function refund() public {
32     require(status == 2 && tBlock), tBlock = true;
33     msg.sender.call(deposits[msg.sender]);//UCL, no RE.
34     deposits[msg.sender] = 0, tBlock = false;
35   }
36   function securedestruct() internal {//Call indirectly.
37     require(msg.sender == owner);
38     selfdestruct(msg.sender); //SU
39   }
40   function destruct(address baddr, uint dvalue) public {
41     Booloracle b = Booloracle(baddr);//Initial contract.
42     require(b.allow(dvalue)); //Fail for current arts.
43     securedestruct();
44   }
45 }
```

Fallback function
Having six public functions
External call
Triggered by four continuous calls
Use mutexes to prevent reentry
Internal call

Listing 1: Example of multiple transaction cooperation.

Why is this detection difficult? As illustrated above, exploitable vulnerabilities are often hidden within a large transition-state space induced by the following contract characteristics, making them likely to be missed. Loops and increasing contract complexity create long or even non-terminating paths [58, 44]; cross-contract calls and fallback functions require preserving caller-callee consistency and Ether-transfer side effects [49, 9, 79]; and multi-transaction exploits require persistent world-state tracking across storage, balances, block metadata, and timestamps [49, 63, 51]. Moreover, constraint solving, path-state storage, taint tracking, and vulnerability checking add substantial overhead, often exhausting time or memory budgets before vulnerable paths are reached [51, 83].

[2]Detailed on `https://github.com/Secbrain/RSymX/tree/main/Vulnerability_examples`.

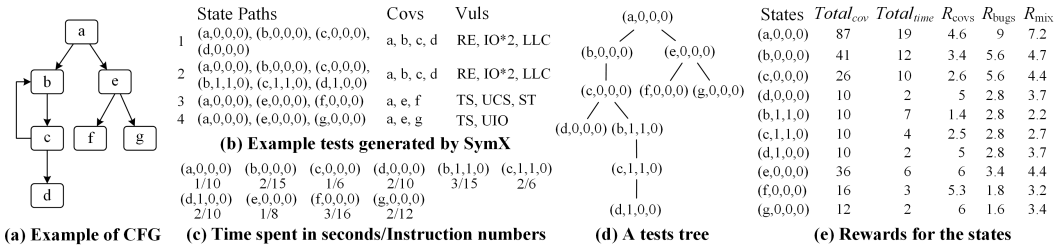


Figure 3: An example of assigning rewards to states explored by SymX.

Why are existing accelerations insufficient? Unguided techniques, such as pruning, heuristic search, and parallelism, improve throughput but still spend substantial budgets on benign states [8, 53, 39, 83, 51]. Existing guided approaches also have limitations: coverage-oriented exploration may over-explore benign regions, whereas vulnerability-oriented exploration may miss deeper exploits [32, 77, 62, 72]. Table 2 further compares representative methods in terms of exploration actions, state features, guidance signals, and execution semantics. In particular, MythrilQL [72] employs Q-learning with sensitive-opcode guidance, while transaction-oriented approaches such as Harvey [78], ConFuzzius [69], and ItyFuzz [61] rely on concrete or hybrid exploration rather than learned ordering of SMT-feasible symbolic successors. Therefore, scalable exploit generation requires a semantics-faithful and budget-aware symbolic executor that prioritizes exploration by both time-normalized coverage and severity-weighted vulnerability yield.

3 RSymX: Learning Strategies for SymX

3.1 Design Overview

Figure 2 depicts the architecture of RSymX, consisting of seven components. Contract Code Inputter generates contract opcodes for vulnerability detection, and Transaction Inputter extracts call information to identify abnormal behaviors. Symbolic Executor builds the contract CFG and explores state paths via EVM stacks. During this process, SMT Constraint Solvers validate path feasibility and solve call parameters based on accumulated constraints. Vulnerability Detectors capture the contextual execution contexts and use the predefined detection logic to detect vulnerabilities, whose correctness can be verified by Vulnerability Validator. Finally, Result Exporter locates defective code statements and exports audit reports. These operations are detailed in SymX [43] or the appendix C. On this basis, RSymX (Reinforced SymX) alleviates the state explosion problem by employing reinforcement learning-based strategies (feature extraction and model inference) to guide Executor in efficiently exploring state paths and discovering bugs quickly within limited paths.

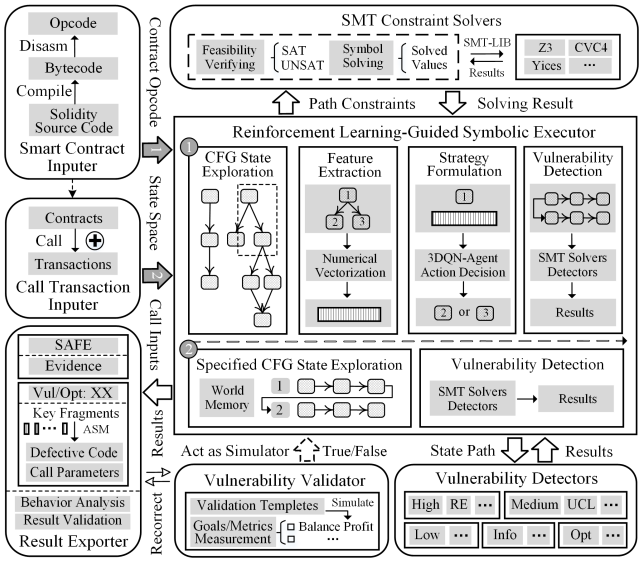


Figure 2: Overview architecture of RSymX.

3.2 Assigning Rewards to Explored States

The reward function guides the symbolic executor to prioritize states that are both useful for code coverage and likely to expose vulnerabilities. RSymX therefore assigns each explored state x_t a hybrid reward consisting of a time-normalized coverage reward and a severity-weighted vulnerability reward. (i) *Code coverage rewards.* Instead of using block coverage [32], RSymX measures instruction-level coverage to distinguish states with different execution costs. Given the exploration tree in Fig. 3, the

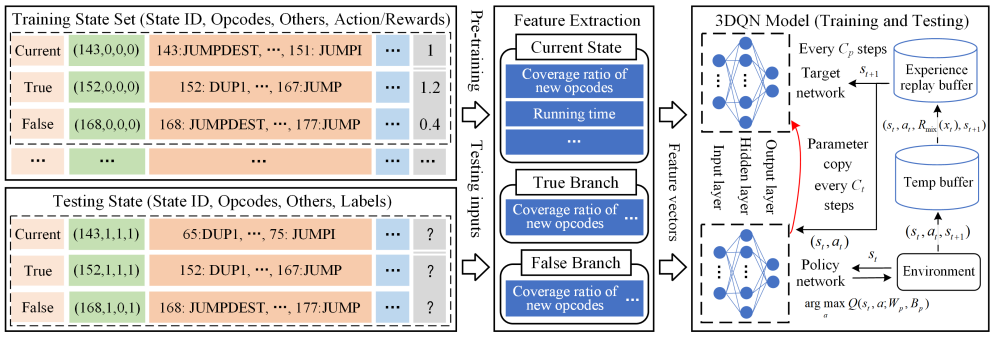


Figure 4: The architecture of the reinforcement learning model. Among them, the column labeled "Others" includes additional information used to calculate the feature vectors, such as execution depth.

cumulative new coverage and execution time of x_t are computed bottom-up:

$$Total_{cov}(x_t) = Newcov(x_t) + \sum_{x \in X_{t+1}} Total_{cov}(x),\ Newcov(x_t) = \begin{cases} \|x_t\| \text{ if } Num_{x_t} == 0 \\ 0 \quad \text{otherwise.} \end{cases}, \quad (1)$$

where $\|x_t\|$ is the instruction numbers of the state x_t. Then, RSymX sums up the time costs.

$$Total_{time}(x_t) = Time(x_t) + \sum_{x \in X_{t+1}} Total_{time}(x),\ Time(x_t) = \sum_{ins \in x_t} Time(ins). \quad (2)$$

The coverage reward is calculated by Eq. (3), where β scales the time-normalized coverage reward. As shown in Fig. 3, the state (d,0,0,0) is worth exploring before (b,1,1,0) given the efficient coverage.

$$R_{\text{covs}}(x_t) = \beta \cdot Total_{cov}(x_t) / Total_{time}(x_t) \quad (3)$$

(ii) *Bug discovery rewards.* To prioritize exploitable and high-impact paths, RSymX weights each detected vulnerability by severity and initialize $\omega = \{\text{High} : 1.0, \text{Medium} : 0.8, \text{Low} : 0.6, \text{Opt} : 0.4, \text{Info} : 0.2\}$. It compute the discovery reward by Eq. (4), where $num_{vul}[\cdot]$ refers to the number of vul in the $path$ when $\cdot$ is true, otherwise 0. In this way, the state (b,0,0,0) is explored before (e,0,0,0).

$$R_{\text{bugs}}(x_t) = \sum_{vul \in vuls} Total_{vul}(x_t) \cdot \omega[\text{Severity}(vul)],\ Total_{vul}(x_t) = \sum_{p \in paths} num_{vul}[x_t \in p], \quad (4)$$

(iii) *Hybrid rewards.* RSymX combines the two objectives as a mixed reward following Eq. (5), where $\aleph \in [0, 1]$ controls the preference between vulnerability discovery and new-state exploration. For example, the state (g,0,0,0) is explored before (f,0,0,0), given the execution efficiency when $\aleph$=0.6.

$$R_{\text{mix}}(x_t) = \aleph R_{\text{bugs}}(x_t) + (1 - \aleph) R_{\text{covs}}(x_t) \quad (5)$$

3.3 Strategy Learning with Agent

During online symbolic execution, the true reward of a state is unavailable because the executor has no global view of unexplored paths. Thus, RSymX trains a 3DQN agent to estimate the mixed reward of candidate successor states and schedule their exploration, as shown in Fig. 4.

(i) *State Features.* Each symbolic state x_t is encoded as a feature vector $s_t \in \mathbb{R}^d$. As summarized in Table 3, the features capture three kinds of signals: *coverage signals* including new/total instruction coverage of the current state and its true/false branches; *execution-cost signals* including constraint size, stack size, running time, and depth; and *transaction-sequence signals* including visit counts and transaction index. These features allow 3DQN

Table 3: Features of states in the exploration process.

Feature	Description
covnumNew	(1) Number of new instructions covered by the current state. (2) Number of new instructions covered by the true branch. (3) Number of new instructions covered by the false branch.
covratioNew	(1) Ratio of new instructions covered by the current state. (2) Ratio of new instructions covered by the true branch. (3) Ratio of new instructions covered by the false branch.
covnum	(1) Number of instructions covered by the current state. (2) Number of instructions covered by the true branch. (3) Number of instructions covered by the false branch.
covratio	(1) Ratio of instructions covered by the current state. (2) Ratio of instructions covered by the true branch. (3) Ratio of instructions covered by the false branch.
visitTimes	(1) Visited times of this state in current transaction. (2) Visited times of state's true branch in current transaction. (3) Visited times of state's false branch in current transaction.
constraint	Size of the current state's last constraints.
stack	Size of the current state's last call stack.
time	Running time of all instructions covered by the current state.
transeq	Transaction serial number of the state execution.
depth	Number of states in current transaction along the path.

to distinguish states with different coverage potential, execution cost, and exploitability, which are detailed in the appendix.

(ii) *Reward estimation.* 3DQN uses a policy network Q and a target network $\hat{Q}$ with the same MLP architecture but different parameters (W_p, B_p) and (W_t, B_t). The policy network ranks successor states, while the target network stabilizes training and mitigates overestimation [25]. Currently, each network is implemented as a three-layer MLP. Taking Q as an example, the hidden layer maps s_t to $h^p \in \mathbb{R}^{1\times q}$, and the output layer predicts the action's reward:

$$h^p = s_t W_h^p + B_h^p, h^t = s_t W_h^t + B_h^t, \tag{6}$$

$$Q(s_t, a; W_p, B_p) = h^p W_o^p + B_o^p = s_t W_p + B_p, \tag{7}$$

$$\hat{Q}(s_t, a; W_t, B_t) = h^t W_o^t + B_o^t = s_t W_t + B_t, \tag{8}$$

where $W_p = W_h^p W_o^p$, $B_p = B_h^p W_o^p + B_o^p$, $W_t = W_h^t W_o^t$ and $B_t = B_h^t W_o^t + B_o^t$. The dimension $v = 2$ corresponds to the two successor states of a symbolic branch. For a symbolic state with two possible jump destinations, 3DQN uses an ε-greedy strategy to determine the exploration order:

$$A_t = \begin{cases} \text{shuffle}([a_t]_{a_t\in[0,v)\wedge a_t\in\mathbb{Z}}) & \text{if probability } \varepsilon \\ \arg\operatorname*{sort}_a Q(s_t, a; W_p, B_p) & \text{otherwise.} \end{cases} \tag{9}$$

The Symbolic Executor then explores $X_{t+1}[a]$ for $a \in A_t$ sequentially under DFS. Thus, states with larger estimated cumulative rewards are explored earlier within the path budget n_{paths}, while all paths can still be explored when the budget is sufficiently large.

(iii) *Iterative learning.* To adapt to changing state distributions across contracts, 3DQN updates the policy network every C_p predictions and periodically synchronizes the target network every C_t predictions. Exploration transitions (s_t, a_t, s_{t+1}) are stored in a temporary buffer D_r and then inserted into the replay buffer D after rewards are available. The target of each sampled transition is:

$$a_{t+1}^p = \arg\max_a Q(s_{t+1}, a; W_p, B_p), y_t = R_{\text{mix}}(x_t) + \gamma \hat{Q}(s_{t+1}, a_{t+1}^p; W_t, B_t) \tag{10}$$

The policy network is optimized by minimizing the Mean Squared Error (MSE) in Eq. (11) via gradient descent [1], where α refers to the learning rate. By using Q to select the next action and $\hat{Q}$ to evaluate it, 3DQN mitigates value overestimation and stabilizes learning. Through iterative updates, it learns to prioritize states with higher coverage gain or more severe vulnerabilities.

$$loss \mathrel{+}= \frac{1}{n}(y_t - Q(s_t, a_t; W_p, B_p))^2, Q(s_t, A_t; W_p, B_p) \mathrel{+}= \alpha(y_t - Q(s_t, A_t; W_p, B_p)) \tag{11}$$

4 Evaluation

4.1 Experiments Setup

Baselines. SOTA methods include pattern matching (Slither [21] and SmartCheck [67]), symbolic execution (Oyente [49], Park-Mythril [83], MythrilQL [72], DefectChecker [9], SmartTest [62], Manticore [51], SymX [43]), fuzz testing (ILF [31], RLF [66], SMARTIAN [13]) and ML (TMP [84]).

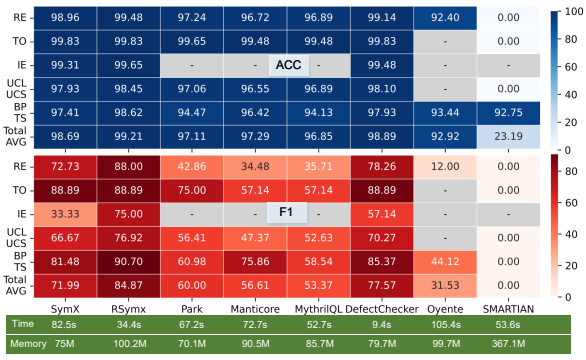


Figure 5: Performance of methods on Dataset$_2$.

Datasets. Dataset$_1$ integrates two open-source datasets SB-Curated [15] and Mini-contracts [41], involving 158 contracts and 1,131 labeled defects, with 411 branches on average and up to 7,396 branches under $TX = 2$. Dataset$_2$ contains 579 bytecodes with 86 file-level issues [9], averaging 3,597 instructions and 30 functions, and holding over 4M Ether in total. Dataset$_3$ includes 13,413 wild source-code contracts, Dataset$_4$ includes 183,710 bytecodes [10], and Dataset$_5$ contains 27 real-world incident contracts.

Metrics. For Dataset$_1$, we use fine-grained vulnerability labels defined by type and code location. A report is TP if both match a labeled item, FP if unmatched, and FN if a labeled item is missed.

Table 4: Detection results on Dataset$_1$. "-" means unsupported projects. P/R refer to the appendix G.

Severity	Metric	SymX	RSymX	SmartCheck	Slither	Park	Manticore	MythrilQL	DefectChecker	SmartTest	Oyente	ILF	RLF	SMARTIAN	TMP
High	OS	90.87	**93.73**	-	82.80	83.26	81.18	81.33	82.61	83.54	67.11	81.58	87.65	73.68	70.10
	F1	95.20	**96.78**	-	90.70	90.68	89.34	89.45	90.48	90.98	79.74	89.74	93.30	84.63	82.61
Medium	OS	89.90	**91.13**	84.07	85.59	77.92	72.66	73.58	83.84	86.94	66.15	68.79	79.32	72.59	67.84
	F1	94.69	**95.36**	91.09	91.97	87.66	84.16	84.84	91.19	93.02	79.63	81.47	88.47	83.90	80.94
Low	OS	90.59	**95.29**	61.90	86.36	73.91	90.59	69.57	82.22	-	69.57	83.77	87.28	90.59	71.65
	F1	95.06	**97.59**	76.47	92.68	85.01	95.06	82.06	90.24	-	82.05	91.15	93.18	95.06	83.61
Info	OS	89.29	**89.88**	66.07	87.21	-	-	-	-	-	-	-	-	-	68.89
	F1	94.34	**94.67**	79.57	93.17	-	-	-	-	-	-	-	-	-	81.17
Opt	OS	89.29	**90.45**	60.00	92.94	-	-	-	-	-	-	-	-	-	76.32
	F1	94.32	**94.93**	75.00	96.34	-	-	-	-	-	-	-	-	-	86.53
Overall	OS	90.22	**92.39**	72.32	85.05	79.62	80.00	76.37	83.36	84.90	67.48	78.93	85.47	76.63	70.20
	F1	94.88	**96.06**	84.03	91.93	88.57	88.64	86.52	90.94	91.81	80.30	88.04	92.09	86.46	82.61

Because exhaustive line-level negatives are unavailable, TN is not computed. We report overlap score (OS), precision (P), recall (R), and F1 [67, 21, 49, 41]. For contract-level Dataset$_2$, TN is available, so we additionally report accuracy (ACC). Also, Time (T) and memory (M) measure efficiency.

Implementation details. All experiments were performed on a PC running Ubuntu 18.04 and equipped with an Intel Core i7-10875H and 8GB of RAM. RSymX is implemented in Python with a 27K lines of code. MLP network is implemented with PyTorch, Adam optimizer [57], and MSE loss function. Also, we employ solc with multiple versions to compile source code, utilize pyevmasm 0.2.3 to disassemble bytecode, and use Z3 v4.12.1.0 to solve symbolic constraints. For training, the policy is warm-started on 300 contracts from Dataset$_3$, disjoint from Dataset$_1$, Dataset$_2$, and the reported case-study contracts. The warm-start uses no ground-truth vulnerability labels, and rewards are derived only from symbolic execution and vulnerability detectors.

Parameter settings. If not specifically emphasized, n_{paths} is set 3,200 and 400 in SymX and RSymX, respectively. Also, the thresholds $TX = 2$ and $R = 2$ are set. For the 3DQN model, $\aleph = 0.6$, $C_p = 800$, $C_t = 1200$, and $C_r = 400$.

4.2 Vulnerability Detection Evaluation

Table 4 reports the results on Dataset$_1$.[3] Overall, RSymX outperforms existing tools across most severities. Its precise execution semantics restore realistic contract paths, while the 3DQN-guided strategy finds vulnerable paths with fewer explorations, alleviating failures caused by state explosion.

Compared with SymX, RSymX improves both effectiveness and efficiency, reducing average analysis time from 68.4s to 30.7s. On the bytecode Dataset$_2$ in Fig. 5, RSymX shows consistent gains, achieving 99.21% ACC and 84.87% F1. Its F1 is 10-20% higher than Park-Mythril/MythrilQL and exceeds DefectChecker (77.57%). This improvement stems from mixed rewards that guide exploration toward both coverage-increasing and bug-revealing paths, balancing precision and recall.

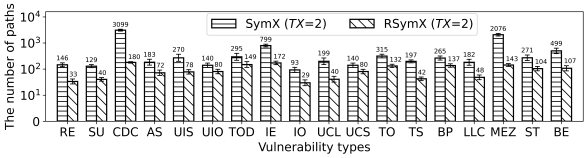


Figure 6: The average number of explored paths.

4.3 Acceleration effectiveness of RSymX

Figure 6 compares the paths needed by SymX and RSymX to find the same bugs. Across most vulnerability types, RSymX reduces paths and speeds up symbolic execution by 2x-4x, and up to 10x in some cases. Under a tighter budget (n_{paths}: 400→200), RSymX still maintains strong performance in Fig. 7, improving recall and keeping precision above 93%.

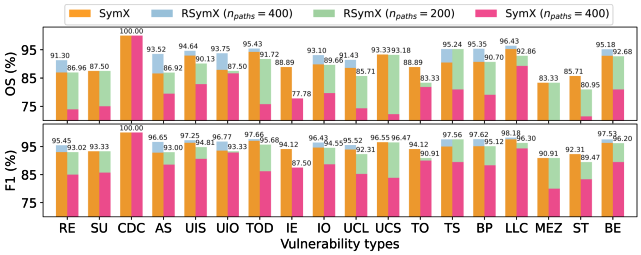
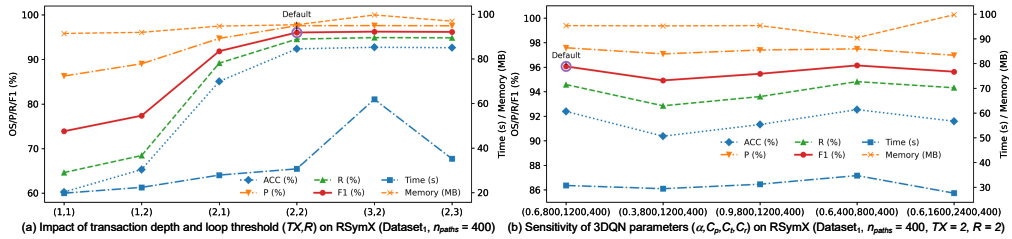


Figure 7: Effects of path limits $n_{paths} \in [400, 200]$.

[3]Details are available at `https://github.com/Secbrain/RSymX/tree/main/Data1`.



(a) Impact of transaction depth and loop threshold (TX,R) on RSymX (Dataset$_1$, n_{paths} = 400) (b) Sensitivity of 3DQN parameters (α,C_p,C_t,C_l) on RSymX (Dataset$_1$, n_{paths} = 400, TX = 2, R = 2)

Figure 8: Effect of limits (transaction depth TX and loop threshold R) and parameters on RSymX.

4.4 Examples of Vulnerable Wild Contracts

We examine representative wild contracts and real security incidents. Table 5 lists active and valuable contracts from Dataset$_2$ correctly detected by RSymX but misreported by other tools.[4] Table 6 further shows reported attack cases from Dataset$_5$. For example, RSymX detects an IO bug in the *SmartMesh* contract, where attackers can manipulate transferProxy parameters so that _fee+_value=0 and bypass validation. It also detects arithmetic-checking flaws in contracts such as *EMVC*. These cases show that RSymX can discover exploitable vulnerabilities from real runtime contract bytecode.

Table 5: Real-world examples. 3rd: number of transactions, capped at 20,000; 4th: contract balance in Wei.

Vul.	Contract address	#Tx	Balance	SymX	RSymX	Other Methods
IO	0x80d0f44bf7...43a006	25	1.6E+15	TP	TP	FNs , *e.g.,* SMARTIAN
	0x43472E1018...8a7f6E	61	6.7E+14	FN	TP	FNs , *e.g.,* MythrilQL
W/o IO	0xb60aeb53a7...938db9	2	1.0E+16	TN	TN	FPs , *e.g.,* SmartTest
	0x2040f2f2bb...7922c1	1,816	1.2E+18	TN	TN	FPs , *e.g.,* Oyente
RE	0xd4260e4bfb3...a5f37a	157	0.0E+18	TP	TP	FNs , *e.g.,* MythrilQL
	0xfeb75b3cc7...9a6e36	14	2.9E+15	TP	TP	FNs , *e.g.,* SmartTest
W/o RE	0x3863fb40d6...30ae6f	24	5.0E+16	TN	TN	FPs , *e.g.,* Slither
	0x81dcc6246f...f3375f	72	0.0E+00	TN	TN	FPs , *e.g.,* Oyente
TOD	0x7a35fe1e97...d7deb4	226	5.6E+16	TP	TP	FNs , *e.g.,* Park
	0x6ffbd6b41b...02f2ab	82	7.0E+16	TP	TP	FNs , *e.g.,* Oyente
UIS	0xf6E435b7e8...070671	9	1.1E+16	TP	TP	FNs , *e.g.,* Slither
	0x2b73cb82ae...72abb6	1,079	5.3E+16	TP	TP	FNs , *e.g.,* Manticore
BP	0x540834aa7a...898538	20	2.5E+16	TP	TP	FNs , *e.g.,* SmartCheck
	0xf3122a43ee...0073e2	43	4.7E+13	TP	TP	FNs , *e.g.,* SMARTIAN

4.5 Ablation and Sensitivity Analysis (RQ4)

Parameters. We further analyze TX, R, and 3DQN scheduler parameters. (i) *Effect of TX and R.* Fig. 8(a) shows that increasing TX from 1 to 2 improves F1 from 73.90/77.40 to 91.87/96.06 with moderate time growth, since many exploits require cross-transaction states. Increasing R from 1 to 2 exposes more loop-dependent branches, but larger settings yield diminishing returns, *e.g.,* TX=3 doubles time to 61.9s while reaching mostly equivalent states. Thus, TX=2 and R=2 balance exploit coverage and exploration cost. (ii) *Effect of 3DQN parameters.* Fig. 8(b) shows stable F1 across $\alpha \in [0.3, 0.9]$ within about $\pm 1\%$, with runtime around 29-31s. Update intervals (C_p, C_t) mainly trade adaptation speed for stability. Overall, the parameters mainly affect efficiency, while TX and R dominate effectiveness by enabling contract semantics.

```
1 mapping (address => uint256) balances;
2 function transferProxy(uint256 _value,
      uint256 _feeSmt,...) public ...{
3   if(balances[_from]<_feeSmt+_value) revert();
4   Transfer(_from, msg.sender, _feeSmt);
5   ...                          Overflow bypass check
6 }
```

Listing 2: SMT contract with IO vulnerability.

Baseline models. Given the modular design, the agent can replace the default MLP while retaining the same state features and mixed reward. Table 7 compares MLP with models such as Bi^2-LSTM/GRU and RF. Most variants outperform SymX,while the performance differences across approximators highlight the importance of model selection. Heavier models bring only small gains: Bi^2-LSTM/GRU improve recall, while XGB gives the best precision-recall

Table 6: Examples of contract events. "-" means unknown.

Vul.	Incidents	Contract address	Money	Loc	SymX	RSymX	Others
RE	The Dao	0x304a554a31...D83490	$60M	201	TP	TP	FNs, *e.g.,* SMARTIAN
BP	FOMO3D	0xA62142888A...389Da1	$3M	1365	TP	TP	FNs, *e.g.,* MythrilQL
ST	KotET	0xb336a86e2F...F254D0	-	105	TP	TP	FNs, *e.g.,* Oyente
TOD	AMORCOIN	0x14Fb4c93Fe...867969	-	32	FN	TP	FNs, *e.g.,* Manticore
IO	EMVC	0xd3F5056D9a...4c6AAA	-	16	TP	TP	FNs, *e.g.,* Oyente
IO	BEC	0xC5d105E637...2F793D	$900M	261	FN	TP	FNs, *e.g.,* Park
IO	SMT	0x43eE79e379...93b644	-	134	TP	TP	FNs, *e.g.,* SmartCheck
IO	AMR	0x96c833e434...29bbb5	-	205	TP	TP	FNs, *e.g.,* Slither
IO	SIGMA	0x03AF370732...fc4fd9	-	1873	FN	TP	FNs, *e.g.,* Manticore
IO	UCN	0x6EF5B9ae72...9dAc42	-	13	TP	TP	FNs, *e.g.,* SMARTIN
IO	ETHX	0x1c98eea5fe...4fd481	-	205	TP	TP	FNs, *e.g.,* TMP
IO	H3H3	0x04129cff7d...79a4de	-	205	FN	TP	FNs, *e.g.,* MythrilQL
IO	NUMB	0x310b15ad0a...b179c8	-	205	TP	TP	FNs, *e.g.,* Park
IO	POSH	0x98307e036c...8b38d0	-	183	TP	TP	FNs, *e.g.,* MythrilQL
IO	POWC	0x0f7524d64c...4795d4	-	205	TP	TP	FNs, *e.g.,* Park
IO	POWH	0xa7ca36f727...728a7a	-	205	TP	TP	FNs, *e.g.,* TMP
IO	POWH	0xe923dD8601...807f7C	-	205	TP	TP	FNs, *e.g.,* Manticore
IO	POWH3	0xc825aa83f1...8a002f	-	181	TP	TP	FNs, *e.g.,* TMP
IO	PWHS	0x9f4fd6c336...b880d7	-	207	TP	TP	FNs, *e.g.,* Manticore

[4]More results are available at `https://github.com/Secbrain/RSymX/tree/main/Data2`.

Table 7: Detection performance (%) of RSymX with different baseline models on Dataset$_1$.

Severity	Metrics	SymX		MLP		Bi^2-LSTM		Bi^2-GRU		CNN		RF		DT		XGB		SVM	
High	OS P	90.87	96.64	93.73	97.91	**94.20**	98.10	94.00	97.80	93.50	97.60	93.60	**98.20**	92.80	96.80	94.10	97.90	93.10	97.10
	R F1	93.81	95.20	95.68	96.78	**96.00**	**96.90**	95.80	96.70	95.40	96.50	95.20	96.40	94.50	95.60	95.90	**96.90**	95.00	95.90
Medium	OS P	89.90	95.39	91.13	95.77	**91.70**	96.00	91.50	95.90	90.90	95.60	91.20	**96.10**	90.20	95.00	91.60	95.90	90.70	95.30
	R F1	94.01	94.69	94.95	95.36	**95.30**	**95.70**	95.10	95.50	94.80	95.20	94.70	95.40	94.20	94.60	95.20	95.60	94.60	95.00
Low	OS P	90.59	100.00	95.29	**100.00**	**96.46**	100.00	95.29	100.00	85.83	100.00	89.37	100.00	81.12	100.00	92.91	100.00	91.75	100.00
	R F1	90.59	95.06	95.29	97.59	**96.46**	**98.20**	95.29	97.59	85.83	92.37	89.37	94.39	81.12	89.58	92.91	96.33	91.75	95.70
Info	OS P	89.29	100.00	89.88	100.00	**90.48**	**100.00**	89.88	100.00	87.50	100.00	88.69	100.00	86.31	100.00	89.88	100.00	88.10	100.00
	R F1	89.29	94.34	89.88	94.67	**90.48**	**95.00**	89.88	94.67	87.50	93.33	88.69	94.01	86.31	92.65	89.88	94.67	88.10	93.67
Opt	OS P	89.29	95.52	90.45	96.13	**91.00**	96.50	90.80	96.30	90.30	96.00	90.60	**96.70**	89.80	95.50	90.90	96.40	90.10	95.90
	R F1	93.15	94.32	93.77	94.93	**94.30**	**95.40**	94.10	95.20	93.80	95.00	93.60	95.10	93.00	94.40	94.20	95.30	93.40	94.80
Overall	OS P	90.22	96.91	92.39	97.58	**92.98**	98.51	92.52	98.41	89.82	98.15	90.92	**98.57**	88.26	97.88	92.09	98.39	90.97	97.98
	R F1	92.93	94.88	**94.58**	96.06	93.89	**96.14**	93.52	95.90	90.85	94.36	91.74	95.06	89.29	93.39	93.17	95.76	92.02	94.91
	Time	68.4s		30.7s		35.2s		34.0s		32.5s		33.8s		31.5s		34.5s		32.0s	
	Memory	70.5M		95.3M		150.2M		138.1M		120.4M		110.7M		82.4M		118.3M		88.6M	

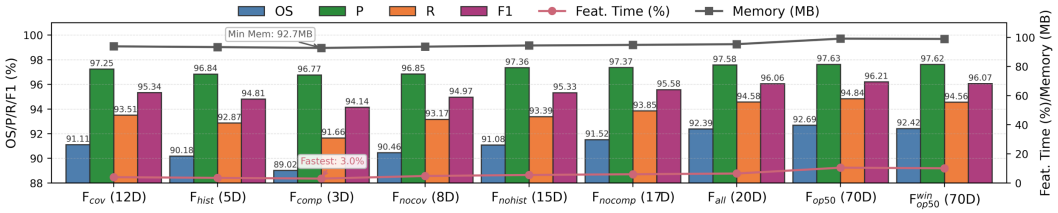


Figure 9: Impact of state feature variants on RSymX (Dataset$_1$, n_{paths}=400, TX=2, and R=2).

trade-off. These marginal gains derive from their characteristics and do not justify the additional computational overhead of more complex models in budget-constrained symbolic execution. Thus, the main improvement comes from the sequence-aware features and mixed rewards, rather than a specific approximator.

State features. RSymX uses a compact 20D state representation combining coverage/progress (F_{cov}, 12D), trajectory/history (F_{hist}, 5D), and complexity/cost (F_{comp}, 3D). Figure 9 shows that using any single group or removing one group degrades F1, *e.g.*, F_{cov} alone drops by 0.72% and F_{nocov} drops by 1.24%. This confirms that coverage drives exploration, history captures sequence dependencies, and complexity makes scheduling budget-aware. Adding 50 opcode features brings marginal gain (96.06→96.21) but higher feature time (6.5%→10.5%) and memory (95.3MB→99.1MB). Windowed opcode aggregation yields a similar F1 (96.07) with comparable overhead. Thus, compact semantic features provide the most useful scheduling signals.

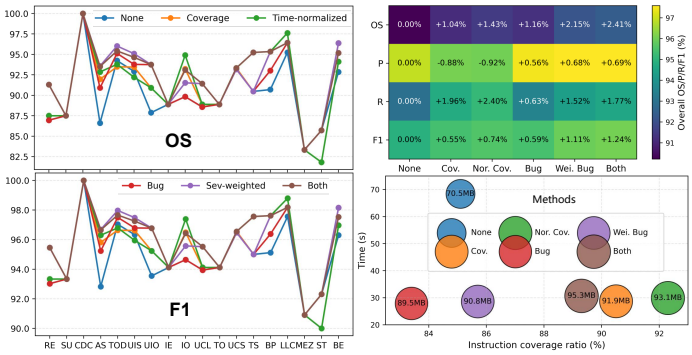


Figure 10: Effects of reward schemes ($n_{\text{paths}} = 400, TX = 2$).

Varying rewards. RSymX combines time-normalized coverage and severity-weighted vulnerability rewards, with effects shown in Fig. 10. The mixed scheme performs best, achieving 92.39% OS and 96.06% F1. It improves F1 by +1.18% over no reward and by +0.47-0.66% over single-objective variants, while maintaining moderate runtime and high instruction coverage. Coverage-only rewards favor recall and coverage, whereas bug-only rewards favor precision but may miss deeper paths. The mixed reward balances them by steering the executor toward unseen behaviors and vulnerable paths.

Reinforcement algorithms. We compare 3DQN with Policy, DQN, A2C, and PPO under the same setting, as shown in Fig. 11. All RL-guided variants outperform SymX, improving F1 from 94.88% to 95.38-96.06% and reducing runtime from 68.4s to 26.6-35.8s. DQN improves stability with a target network and a replay buffer. A2C and PPO also help, but are slower in this discrete-reward space. These results show that RSymX is compatible with multiple RL algorithms, and 3DQN further stabilizes bootstrapping via delayed updates and achieves the best effectiveness-efficiency balance.

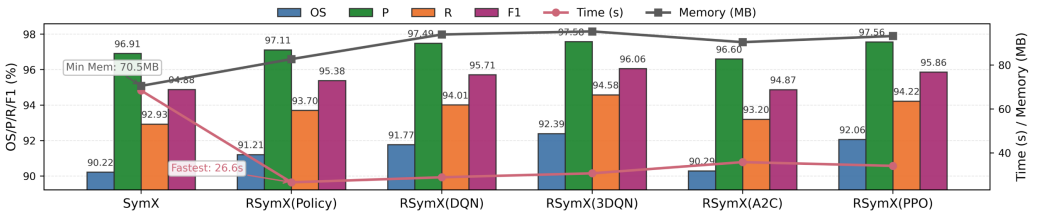


Figure 11: Effectiveness of various RL schedulers on Dataset$_1$ ($n_{\text{paths}=400}$, $TX=2$, $R=2$).

Scheduler attribution. Under the same 400-path budget and fixed SymX execution semantics, the non-learned hybrid greedy scheduler achieves 95.69% F1, compared with 93.98% for DFS and 96.06% for RSymX. This indicates that the hybrid guidance accounts for most of the improvement, while RL-based delayed credit assignment provides a smaller additional gain, especially on multi-transaction and deep-path contracts.

5 Discussion

Feature-aware insights. We analyze the random-forest agent with global feature importance and SHAP explanations to check whether the learned policy is feature-aware. Dominant features include branch-local coverage, the transaction index, and stack context, indicating that the scheduler primarily responds to nearby novelty and transaction-dependent states. This matches the mixed reward: coverage drives short novel paths, while vulnerability rewards sharpen decisions near high-risk states.

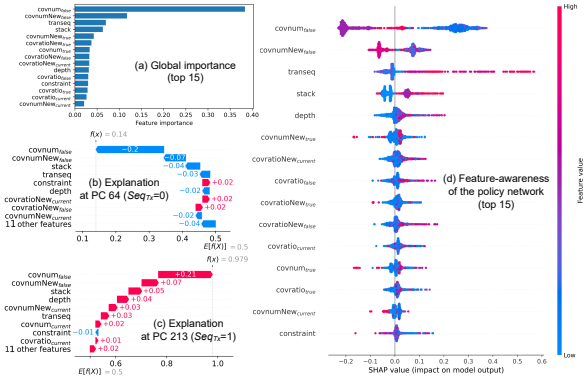


Figure 12: Feature-awareness of the policy network.

Practical implications and extensibility. By building on the semantics-faithful execution and vulnerability validation of SymX, RSymX focuses on improving exploration efficiency without modifying the underlying execution semantics. Its modular scheduling design offers the potential for integration with other symbolic execution frameworks. Moreover, the evaluations on labeled benchmarks and large-scale real-world contracts demonstrate its practical value for prioritizing high-impact vulnerability discovery under limited analysis budgets.

Limitations and improvements. RSymX prioritizes vulnerable paths under a fixed budget, but safe contracts may still require broad exploration. It may also miss vulnerabilities beyond n_{paths} or time limits, especially in long paths. Future work will explore loop summarization, adaptive solver timeouts, and ABI-aware call-buffer construction. Detailed discussion refers to Appendix H.

6 Conclusion and Future Work

We presented RSymX, a reinforcement-learning-guided symbolic execution framework built on SymX for efficient and exploitable smart-contract vulnerability detection. By using mixed coverage and vulnerability rewards, RSymX prioritizes high-yield states and mitigates state explosion in large contracts. Extensive experiments show that RSymX improves both detection effectiveness and exploration efficiency over existing methods. Future work will integrate RSymX's scheduling strategy into other analyzers and further improve scalability for complex contract behaviors.

Acknowledgment. The work was supported by the National Key R&D Program of China under Grant No. 2023YFB2705000, the Young Scientists Fund of the National Natural Science Foundation of China (NSFC) under Grant No. 62602666, the China Postdoctoral Science Foundation under Grant No. GZB20260393, and the fund of Laboratory for Advanced Computing and Intelligence Engineering.

References

[1] Marcin Andrychowicz et al. Learning to learn by gradient descent by gradient descent. In *NIPS*, pages 3981–3989, 2016.

[2] Clark W. Barrett, Christopher L. Conway, Morgan Deters, et al. CVC4. In *CAV*, volume 6806, pages 171–177. Springer, 2011.

[3] Clark W. Barrett et al. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[4] Karthikeyan Bhargavan et al. Formal verification of smart contracts: Short paper. In *PLAS@CCS*, pages 91–96. ACM, 2016.

[5] Priyanka Bose et al. SAILFISH: vetting smart contract state-inconsistency bugs in seconds. In *IEEE S&P*. IEEE, 2022.

[6] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, pages 183–198. ACM, 2011.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.

[8] Sooyoung Cha and Hakjoo Oh. Making symbolic execution promising by learning aggressive state-pruning strategy. In *ESEC/SIGSOFT FSE*, pages 147–158. ACM, 2020.

[9] Jiachi Chen, Xin Xia, et al. DEFECTCHECKER: automated smart contract defect detection by analyzing EVM bytecode. *IEEE Trans. Software Eng.*, 2020.

[10] Jiachi Chen, Xin Xia, et al. Defining smart contract defects on ethereum. *IEEE Trans. Software Eng.*, 48(2):327–345, 2022.

[11] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, et al. SODA: A generic online detection framework for smart contracts. In *NDSS*. The Internet Society, 2020.

[12] Ting Chen et al. Cloud-based parallel concolic execution. In *SANER*, pages 437–441. IEEE Computer Society, 2017.

[13] Jaeseung Choi, Doyeon Kim, et al. SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *ASE*, pages 227–239. IEEE, 2021.

[14] Leonardo Mendonça de Moura et al. Z3: an efficient SMT solver. In *TACAS*, volume 4963, pages 337–340. Springer, 2008.

[15] Thomas Durieux, João F. Ferreira, et al. Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. In *ICSE*, pages 530–541. ACM, 2020.

[16] Bruno Dutertre. Yices 2.2. In *CAV*, volume 8559, pages 737–744. Springer, 2014.

[17] Ehervm. Ethereum virtual machine opcodes. [EB/OL], 2024. `https://ethervm.io/` Accessed March 1, 2024.

[18] Ethereum. The solidity language. [EB/OL], 2020. `https://github.com/ethereum/solidity` Accessed May 1, 2021.

[19] Ethereum. Go-ethereum. [EB/OL], 2022. `https://github.com/ethereum/go-ethereum` Accessed May 1, 2022.

[20] ethereum.org. Ethereum virtual machine (evm). [EB/OL], 2024. `https://ethereum.org/en/developers/docs/evm/`. Accessed April 1, 2024.

[21] Josselin Feist et al. Slither: a static analysis framework for smart contracts. In *WETSEB@ICSE*, pages 8–15. IEEE / ACM, 2019.

[22] Peixuan Feng, Yongjuan Wang, et al. Evaluation framework for smart contract fuzzers. *J. Softw. Evol. Process.*, 37(4), 2025.

[23] João F. Ferreira et al. Smartbugs: A framework to analyze solidity smart contracts. In *ASE*, pages 1349–1352. IEEE, 2020.

[24] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *USENIX Security Symposium*, pages 2757–2774. USENIX Association, 2020.

[25] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *ICML*, volume 80, pages 1582–1591. PMLR, 2018.

[26] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, et al. Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding. In *ICSME*, pages 394–397. IEEE, 2019.

[27] Gustavo Grieco et al. Echidna: effective, usable, and fast fuzzing for smart contracts. In *ISSTA*, pages 557–560. ACM, 2020.

[28] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *POST*, volume 10804, pages 243–269. Springer, 2018.

[29] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, et al. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL): 48:1–48:28, 2018.

[30] Kazuki Hara, Takeshi Takahashi, et al. Machine-learning approach using solidity bytecode for smart-contract honeypot detection in the ethereum. In *QRS Companion*, pages 652–659. IEEE, 2021.

[31] Jingxuan He, Mislav Balunovic, et al. Learning to fuzz from symbolic execution with application to smart contracts. In *CCS*, pages 531–548. ACM, 2019.

[32] Jingxuan He et al. Learning to explore paths for symbolic execution. In *CCS*, pages 2526–2540. ACM, 2021.

[33] Ningyu He, Ruiyi Zhang, Haoyu Wang, et al. EOSAFE: security analysis of EOSIO smart contracts. In *USENIX Security Symposium*, pages 1271–1288. USENIX Association, 2021.

[34] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, et al. KEVM: A complete formal semantics of the ethereum virtual machine. In *CSF*, pages 204–217. IEEE Computer Society, 2018.

[35] Jianjun Huang, Songming Han, Wei You, et al. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Trans. Inf. Forensics Secur.*, 16:2144–2156, 2021.

[36] Bo Jiang, Ye Liu, et al. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *ASE*, pages 259–269. ACM, 2018.

[37] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, et al. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *IEEE S&P*, pages 1695–1712. IEEE, 2020.

[38] Sukrit Kalra, Seep Goel, et al. ZEUS: analyzing safety of smart contracts. In *NDSS*. The Internet Society, 2018.

[39] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*, pages 1317–1333. USENIX Association, 2018.

[40] You Li, Zhendong Su, et al. Steering symbolic execution to less traveled paths. In *OOPSLA*, pages 19–32. ACM, 2013.

[41] Zhaoxuan Li, Siqi Lu, et al. Smartfast: an accurate and robust formal analysis tool for ethereum smart contracts. *Empir. Softw. Eng.*, 27(7):197, 2022.

[42] Zhaoxuan Li, Siqi Lu, Rui Zhang, et al. Vulhunter: Hunting vulnerable smart contracts at EVM bytecode-level via multiple instance learning. *IEEE Trans. Software Eng.*, 49(11):4886–4916, 2023.

[43] Zhaoxuan Li, Ziming Zhao, Wenhao Li, Rui Zhang, Rui Xue, Siqi Lu, and Fan Zhang. Demo: Enhancing smart contract security comprehensively through dynamic symbolic execution. In *CCS*, pages 5072–5074. ACM, 2024.

[44] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, et al. Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts. *IEEE Trans. Software Eng.*, 49(2):777–801, 2023.

[45] Han Liu et al. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *ASE*, pages 814–819. ACM, 2018.

[46] Zhenguang Liu, Peng Qian, Xiang Wang, et al. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In *IJCAI*, pages 2751–2759, 2021.

[47] Zhenguang Liu, Peng Qian, Xiaoyang Wang, et al. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Trans. Knowl. Data Eng.*, 2021.

[48] Ning Lu, Bin Wang, Yongxin Zhang, et al. Neucheck: A more practical ethereum smart contract security analysis tool. *Softw. Pract. Exp.*, 51(10):2065–2084, 2021.

[49] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, et al. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016.

[50] STEVE MARX. Eunderstanding ethereum smart contract storage. [EB/OL], 2018. `https://programtheblockchain.com/posts/2018/03/09/understanding-ethereum-smart-contract-storage`. Accessed April 1, 2024.

[51] Mark Mossberg et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *ASE*, pages 1186–1189. IEEE, 2019.

[52] Tai D. Nguyen et al. sfuzz: an efficient adaptive fuzzer for solidity smart contracts. In *ICSE*, pages 778–788. ACM, 2020.

[53] Ivica Nikolic et al. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC*, pages 653–663. ACM, 2018.

[54] Hristina Palikareva et al. Multi-solver support in symbolic execution. In *CAV*, volume 8044, pages 53–68. Springer, 2013.

[55] Daejun Park et al. A formal verification tool for ethereum VM bytecode. In *ESEC/SIGSOFT FSE*, pages 912–915. ACM, 2018.

[56] Anton Permenev et al. Verx: Safety verification of smart contracts. In *IEEE S&P*, pages 1661–1677. IEEE, 2020.

[57] PyTorch. Optim. [EB/OL], 2022. `https://pytorch.org/docs/stable/optim.html` Accessed January 1, 2023.

[58] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. Ef/cf: High performance smart contract fuzzing for exploit generation. In *EuroS&P*, pages 449–471. IEEE, 2023.

[59] Michael Rodler et al. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *NDSS*. The Internet Society, 2019.

[60] Christoph Sendner, Lukas Petzi, et al. Large-scale study of vulnerability scanners for ethereum smart contracts. In *IEEE S&P*, pages 220–220. IEEE, 2024.

[61] Chaofan Shou, Shangyin Tan, and Koushik Sen. Ityfuzz: Snapshot-based fuzzer for smart contract. In *ISSTA*, pages 322–333. ACM, 2023.

[62] Sunbeom So et al. Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In *USENIX Security Symposium*, pages 1361–1378. USENIX Association, 2021.

[63] ConsenSys Software. Security analysis tool for evm bytecode. [EB/OL], 2020. `https://github.com/ConsenSys/mythril` Accessed May 1, 2021.

[64] ETH Zurich SRI Lab. Securify v2.0. [EB/OL], 2020. `https://github.com/eth-sri/securify2` Accessed May 1, 2021.

[65] Matt Staats and Corina S. Pasareanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194. ACM, 2010.

[66] Jianzhong Su et al. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In *ASE*, pages 36:1–36:12. ACM, 2022.

[67] Sergei Tikhomirov et al. Smartcheck: Static analysis of ethereum smart contracts. In *WETSEB@ICSE*, pages 9–16. ACM, 2018.

[68] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *USENIX Security Symposium*, pages 1591–1607. USENIX Association, 2019.

[69] Christof Ferreira Torres, Antonio Ken Iannillo, et al. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *EuroS&P*, pages 103–119. IEEE, 2021.

[70] Christof Ferreira Torres et al. Osiris: Hunting for integer bugs in ethereum smart contracts. In *ACSAC*, pages 664–676. ACM, 2018.

[71] Petar Tsankov et al. Securify: Practical security analysis of smart contracts. In *CCS*, pages 67–82. ACM, 2018.

[72] Meng Wang, Weiliang Fei, et al. Reinforcement learning guided symbolic execution for ethereum smart contracts. In *APSEC*, pages 91–100. IEEE, 2023.

[73] Wei Wang, Jingjing Song, Guangquan Xu, et al. ContractWard: Automated vulnerability detection models for ethereum smart contracts. *IEEE Trans. Netw. Sci. Eng.*, 8(2):1133–1144, 2021.

[74] Xinming Wang et al. Contractguard: Defend ethereum smart contracts with embedded intrusion detection. *IEEE TSC*, 13(2):314–328, 2020.

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

[76] Hongjun Wu, Zhuo Zhang, Shangwen Wang, et al. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *ISSRE*, pages 378–389. IEEE, 2021.

[77] Jie Wu et al. Reinforcement learning guided symbolic execution. In *SANER*, pages 662–663. IEEE, 2020.

[78] Valentin Wüstholz and Maria Christakis. Harvey: a greybox fuzzer for smart contracts. In *ESEC/SIGSOFT FSE*, pages 1398–1409. ACM, 2020.

[79] Yinxing Xue et al. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *ASE*, pages 1029–1040. IEEE, 2020.

[80] Yu Yan, Siqi Lu, et al. Has the two-decade-old prophecy come true? artificial bad intelligence triggered by merely a single-bit flip in large language models. In *WWW*, pages 2614–2625. ACM, 2026.

[81] Xingxin Yu, Haoyue Zhao, Botao Hou, Zonghao Ying, and Bin Wu. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In *IJCNN*, pages 1–8. IEEE, 2021.

[82] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven T. Stewart, and Vijay Ganesh. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In *ISSRE*, pages 456–462. IEEE, 2019.

[83] Peilin Zheng et al. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *ISSTA*, pages 740–751. ACM, 2022.

[84] Yuan Zhuang, Zhenguang Liu, et al. Smart contract vulnerability detection using graph neural network. In *IJCAI*, pages 3283–3290. ijcai.org, 2020.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction accurately summarize the paper's main scope and contributions: RSymX uses reinforcement-learning-guided symbolic execution to balance code coverage and vulnerability discovery, generate exploit call sequences, and improve exploration efficiency. The claimed overlap score and acceleration results are supported by the evaluation on Dataset_1, Dataset_2, and the acceleration experiments in Section 4.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses limitations in Section 5 and Appendix H. It notes that RSymX prioritizes vulnerable paths under a fixed budget but secure contracts may still require broad exploration, and that long paths or deeply bounded loops beyond n_{paths} or time limits remain challenging. The paper also outlines future improvements, including loop summarization, adaptive solver timeouts, and ABI-aware call-buffer construction.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not present theoretical results requiring formal assumptions or proofs. The equations define the reward functions, state scheduling policy, and 3DQN update procedure used by RSymX, and their empirical effectiveness is evaluated experimentally.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Section 4.1 describes the baselines, implementation environment, datasets, evaluation metrics, and default parameter settings used in the main experiments. The paper also reports time and memory overheads, provides supplementary experiment details in the appendix, and releases code/data at the anonymized repository link (`https://github.com/Secbrain/RSymX`) stated in the abstract.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The paper provides an anonymized GitHub repository containing the code, datasets, and experiment details needed to reproduce the main results. The abstract states the repository link (`https://github.com/Secbrain/RSymX`), and the appendix provides additional dataset, metric, and experimental result details.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4.1 specifies the datasets, baselines, implementation environment, optimizer, evaluation metrics, and default hyperparameters, including n_{paths}, TX, R, $\aleph$, C_p, C_t, and C_r. Additional training, metric, and dataset details are provided in the appendix and the released repository (`https://github.com/Secbrain/RSymX`).

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: The paper reports deterministic evaluation metrics over fixed benchmark datasets and does not include error bars or statistical significance tests. Instead, it provides detailed per-severity results, ablation studies, sensitivity analyses, and efficiency measurements to support the empirical claims.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Section 4.1 reports the experimental environment, including the CPU, RAM, operating system, implementation language, compiler/disassembler versions, and SMT solver version. The evaluation also reports time and memory overheads for the main tools, enabling readers to understand the computational resources needed to reproduce the experiments.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The research conforms, in every respect, with the NeurIPS Code of Ethics.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: There is no societal impact of the work performed.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The paper cites the original sources of the datasets, baselines, analyzers, compilers, and solvers used in the experiments, including SB-Curated, Mini-contracts, DefectChecker, SymX, Z3, solc, and pyevmasm. The released repository provides access details for the code and data used to reproduce the experiments.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [N/A]

Justification: The paper does not release new assets.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLM is only used for polishing the paper language.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.

Table 8: Qualitative comparison of our methods and existing smart contract analysis approaches. Among them, the capability columns "Exp.", "Loc.", "Sim.", "Val.", "Beh.", and "Eff." refer to (finding) exploitable, (bug) location, simulation (run), (result) validation, behavior (identification), and (analysis) efficiency, respectively.

Method Name	Required Input	Technology	Exp.	Loc.	Sim.	Val.	Beh.	Eff.
SmartCheck [67]	Source code	Pattern matching	No	Yes	No	No	No	High
Slither [21]	Source code	Pattern matching	No	Yes	No	No	No	High
NeuCheck [48]	Source code	Pattern matching	No	Yes	No	No	No	High
Zeus [38]	Source code	Pattern matching	No	Yes	No	No	No	High
SMARTEMBED [26]	Source code	Similarity code matching	No	Yes	No	No	No	High
TMP [84]	Source code	ML (GNN)	No	No	No	No	No	High
Peculiar [76]	Source code	ML (Pre-training)	No	No	No	No	No	High
DeeSCVHunter [81]	Source code	ML (Integration framework)	No	No	No	No	No	High
S-gram [45]	Source code	ML (N-gram)	No	No	No	No	No	High
Oyente [49]	Source code and bytecode	Symbolic execution	Yes	Yes	No	No	No	Low
Manticore [51]	Source code and bytecode	Symbolic execution+fork parallel	Yes	Yes	Yes	No	No	Low
Securify [71]	Source code and bytecode	Pattern matching	No	Yes	No	No	No	High
Mythril [63]	Source code and bytecode	Symbolic execution+path pruning	Yes	Yes	Yes	No	No	Low
MythrilQL [72]	Source code and bytecode	Symbolic execution+ML	Yes	Yes	Yes	No	No	Medium
Park [83]	Source code and bytecode	Symbolic execution+fork parallel	Yes	Yes	Yes	No	No	Medium
DefectChecker [9]	Source code and bytecode	Symbolic execution (w/o solving)	No	Yes	No	No	No	High
Eosafe [33]	Bytecode	Symbolic execution	Yes	Yes	Yes	No	No	Low
Honeybadger [68]	Bytecode	Symbolic execution	Yes	Yes	Yes	No	No	Low
teEther [39]	Bytecode	Symbolic execution	Yes	Yes	Yes	No	No	Low
Sailfish [5]	Bytecode	Symbolic execution	Yes	Yes	Yes	No	No	Low
SmartTest [62]	Bytecode	Symbolic execution	Yes	Yes	Yes	No	No	Low
Contractfuzzer [36]	Bytecode+ABI	Fuzzing	Yes	No	Yes	No	No	Low
ILF [31]	Bytecode+transaction	Fuzzing+ML	Yes	No	Yes	No	No	Low
RLF [66]	Bytecode+transaction	Fuzzing+ML	Yes	No	Yes	No	No	Low
SMARTIAN [13]	Bytecode	Fuzzing+flow analysis	Yes	No	Yes	No	No	Low
EF/CF [58]	Bytecode	Fuzzing	Yes	No	Yes	No	No	Low
Contractembed [35]	Bytecode	ML (Graph embedding)	No	No	No	No	No	High
ContractWard [73]	Source code and bytecode	ML (XGBoost etc.)	No	No	No	No	No	High
ContractGuard [74]	Bytecode+transaction	Path matching	Yes	No	No	No	Yes	High
SODA [11]	Bytecode+transaction	Behavior pattern matching	Yes	No	Yes	No	Yes	High
ECFChecker [29]	Bytecode+transaction	Runtime verification	Yes	No	Yes	No	Yes	Medium
Sereum [59]	Bytecode+transaction	Runtime taint tracking	Yes	No	Yes	No	Yes	Medium
MAIAN [53]	Source code and bytecode	Symbolic execution	Yes	Yes	Yes	Yes	No	Low
SymX (Ours)	**Source code, bytecode,**	**Symbolic execution+parallel**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Low**
RSymX (Ours)	**opcode and transaction**	**+ML(coverage+findings)**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Medium**

A Related Work

Existing contract analysis arts based on pattern matching [21, 67, 38, 48, 71, 64], machine learning [84, 46, 47, 26, 45, 76, 81, 73, 42, 30, 35], and theorem proving [4, 37, 28, 34, 55] can identify vulnerabilities but usually produce the results and snippets, making them difficult to reproduce.

To this end, as listed in Table 8, symbolic execution [49, 70, 9, 24, 68, 56, 62, 79, 5, 43] and fuzzing [52, 27] based methods attempt to generate actual exploit transaction sequences. However, they are difficult to make the expected performance due to the state explosion problem, especially for large contracts [60]. To filter explored paths, Mythril [63], Mpro [82], and MAIAN [53] introduce path-pruning strategies, while Park [83], Cloud9 [6], and ParallelJPF [65] parallelize execution. Nevertheless, they still cannot work well with massive contract states. Furthermore, code coverage (Learch [32] and ILF [31]) and vulnerability discovery (SmarTest [62], MythrilQL [72], TeEther [39], and RLF [66]) based methods prioritize suspicious paths to reduce exploration space. Nonetheless, they still face challenges in balancing code coverage and vulnerability discovery, which can lead to local optima and suboptimal guidance [66].

In summary, existing approaches remain limited in efficiently generating exploit transaction sequences against the state explosion problem. To this end, RSymX employs a dynamic agent with mixed rewards that account for the code coverage and vulnerability discoveries to guide state exploration, enabling more complete and faster analysis.

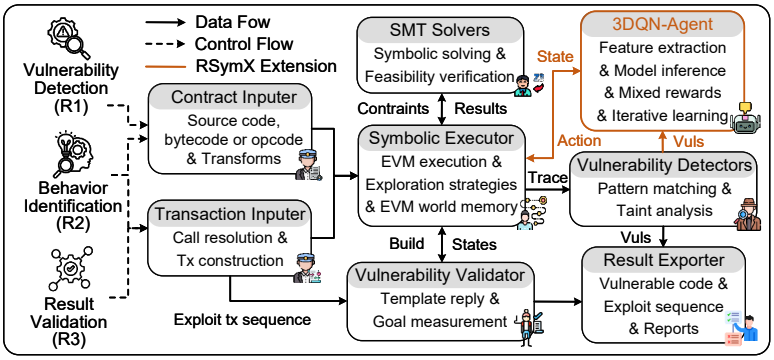


Figure 13: Overview of SymX and its RSymX extension, with the 3DQN-Agent highlighted.

B Detailed Motivation of Symbolic Execution Acceleration

As smart contracts proliferate, the number of deployed contracts on Ethereum reached 66,323,262 by June 2024,[5] as shown in Fig. 1(a). This scale demands vulnerability-analysis techniques that remain effective under severe state explosion. Given a contract set C, each contract $c \in C$ induces a large transition space $X \leftarrow \xi(c)$ over branches, loops, function calls, and transaction-dependent states. The goal is to efficiently discover vulnerable paths $paths_{vuls}$ and their triggering transaction parameters $calls_{vuls}$ from this massive search space.

Why does state/path explosion cripple existing tools? Symbolic execution (*e.g.,* Oyente [49] and Manticore [51]) and fuzzing (*e.g.,* ContractFuzzer [36] and Echidna [27]) can generate replayable exploit sequences, whereas static analyzers (*e.g.,* Slither [21] and ZEUS [38]) and purely learning-based detectors (*e.g.,* TMP [84] and Peculiar [76]) usually report suspicious code without producing concrete exploits. However, Ethereum-specific execution semantics make practical exploration extremely expensive.

(i) Loops and increasing contract complexity. Modern contracts encode increasingly rich business logic and therefore contain more branches, state-changing functions, and loops [58]. Loops enlarge path counts far beyond branch counts and often lead to long or even non-terminating executions. On 78,642 unique contracts filtered from 7.6M open-sourced Ethereum contracts [44], most already have $10^2 \sim 10^3$ symbolic paths, while some have tens of thousands, as shown in Fig. 1(b). Existing tools either time out on deep paths or truncate loops aggressively [49, 39, 62], thereby sacrificing completeness.

(ii) Cross-contract calls and fallbacks. Realistic execution requires expanding paths through inter-contract calls and preserving consistency between caller and callee states. For example, the path of `destruct()` in Listing 1 depends on the branch outcome of `Booloracle.allow(uint)`. Fallback functions triggered during Ether transfer further enlarge the search space and are often unsupported or only approximated by prior tools [49, 9, 79]. Such approximations can mis-handle return values and state updates, causing both false positives and false negatives. Therefore, faithful symbolic execution must explicitly model cross-contract invocations and fallback behaviors.

(iii) Multi-transaction exploit sequences. Many vulnerabilities are exposed only through carefully ordered transaction sequences rather than single calls, as shown in § 2.2. Tracking such behaviors requires persistent world-state management across transactions, including contract storage, balances, block metadata, and timestamps. Prior tools often restrict transaction length or simplify state handling [49, 63, 51], which limits their ability to discover exploits that depend on state accumulation.

(iv) Additional analysis overhead. Beyond EVM execution itself, symbolic analyzers incur substantial cost in constraint solving, path-state storage, taint tracking, and vulnerability checking [51, 83]. These costs grow rapidly with longer paths and richer analyses, often exhausting time or memory budgets before useful results are produced. Therefore, improving solver or detector efficiency alone is insufficient; exploration itself must become more selective.

[5] Data from `https://etherscan.io/chart/deployed-contracts`.

Why are classic unguided accelerations still limited? Prior tools improve throughput through pruning, state deduplication, heuristic search, and parallelism [8, 53, 39, 83, 51]. Although these techniques are helpful, they do not fundamentally solve the problem, because realistic contract analysis still produces far more states than bounded CPU budgets can process.

Why do existing guided methods not work well for symbolic execution? Existing guided approaches suffer from four major limitations. (i) They are mostly fuzzing-oriented rather than state-aware. Tools such as ILF [31], RLF [66], and SMARTIAN [13] mainly focus on seed mutation and transaction generation, but do not reason deeply about symbolic constraints, inter-contract dependencies, or persistent world states. As a result, they often fail to satisfy the precise preconditions required for replayable exploits.

(ii) Single-objective guidance causes bias. Coverage-guided approaches such as Learch [32] and Q-KLEE [77] may over-invest in benign but expensive regions because vulnerable paths are sparse [15, 12]. Conversely, vulnerability-only guidance such as SmarTest [62] and MythrilQL [72] may overvalue suspicious instructions or low-impact findings and become trapped in local optima. In practice, effective guidance should balance *time-normalized coverage* and *severity-weighted vulnerability yield*.

(iii) State features often ignore contract-specific semantics. Prior approaches usually encode local instructions or CFG patterns, but omit contract-specific signals such as transaction order, cross-contract parameters, fallback behaviors, runtime cost, and intermediate world states. Without such semantics, states with very different exploitability may appear similar to the guidance model.

(iv) The learning models are often unstable or weakly adaptive. Many prior studies rely on tabular Q-learning or single-network DQN trained offline [77, 72, 32]. These designs are prone to overestimation and instability under function approximation [25, 80], and they generalize poorly as the exploration distribution shifts during contract analysis.

In summary, generating replayable exploit sequences at scale requires a symbolic executor that is both semantics-faithful and budget-aware. It should prioritize exploration based on both time-normalized coverage and severity-weighted vulnerability yield within limited analysis budgets.

C The Components of SymX

In this section, we supplement the description of SymX. These contents are excerpted from [43], as they are not contributions of this paper. In addition, the main reasons for selecting this tool as our base engine are as follows. (i) This tool has so many functions, *i.e.,* abnormal behavior identification and result validation in addition to vulnerability detection. (ii) The code of this tool was open-sourced, and it can be second-developed easily. (iii) As [43] detailed, this tool embraced a superior performance compared with the other symbolic execution-based tools. (iv) This tool was released recently and is being maintained, so that our questions can be addressed in a timely manner.

C.1 Contract Code/Transaction Inputer

As shown in the left part of Fig. 2, Contract Inputter can feed the source code, bytecode, and opcode of contracts as input. Specifically, because the subsequent contract symbolic execution is performed directly on opcodes, the source code needs to be compiled into bytecode and then disassembled into opcodes using the Geth API [19]. This process can be vividly depicted in Fig. 14. In particular, the compiler can resolve multiple associated contracts uniformly to generate their bytecode. Meanwhile, the assembly language source code file (ASM) can be obtained during the contract compiling and further used to locate the defective source codes based on vulnerable bytecodes/opcodes (*c.f.,* § C.8). In addition, transactions related to the contract calls and the corresponding contract code (either of the above three types, given that they can be transformed into each other) can be used to identify abnormal contract behaviors (§ C.6).

C.2 CFG Construction and State Exploration

Symbol Executor constructs the CFG of contracts based on their opcodes to explore the state transitions during the actual execution. Specifically, the opcodes will be split into several blocks, and their connections will be constructed based on the EVM stack [43, 42, 51]. As an example

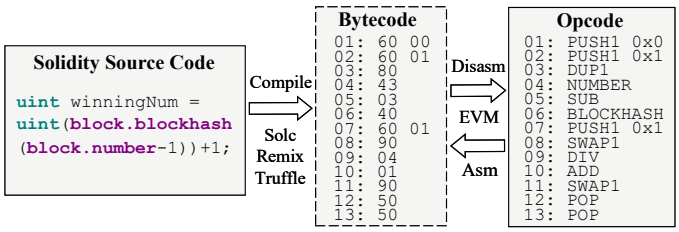


Figure 14: The relationships among source code, bytecode and opcode.

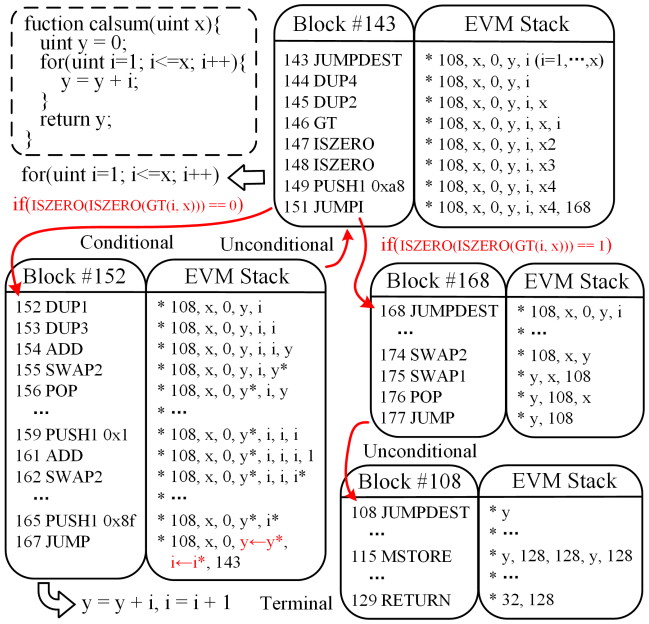


Figure 15: Example of CFG construction and state exploration by Symbol Executor based on symbolic reasoning. This contract code will cause most current symbolic execution-based arts to fall into infinite loops.

shown in Fig. 15, there are 4 blocks that perform the main recursive summation process through a for-loop operation. The instructions in Block #143 represent the conditional judgment expression "i<=x". Block #152 refers to the loop body that executes the addition operation, *i.e.,* "y=y+i" and "i++". Blocks #168 and #108 return the final sum result "y" to the environment. Among them, the leftmost number in each line indicates the instruction index ID, the middle part is the instruction that needs to be reasoned, and the right part shows the runtime values in the EVM stack. During the reasoning process, all instructions in each block will be executed sequentially according to their operations[6], and there is a Program Counter (PC) tracking the visited ID in real-time. Specifically, the PC starts from ID 143 in Block #143 and ends at ID 129 in Block #108. Before the EVM executes the JUMPDEST instruction, there are already some values in the EVM stack, including the input parameter "x", current sum result "y", and iteration variable "i" ($\in [1, x]$). Also, it involves the unconditional jump destination of Block #168, which can be read by the JUMP instruction (ID 177). This is one of the CFG construction advantages by symbolic reasoning, *i.e.,* it can accurately obtain the complex and implicit destinations for jumps without immediate target offsets, unlike jumps at IDs 149-151 and 165-167.

In addition, the conditioned jump by the JUMPI instruction (reading a jump position and a conditional expression from the EVM stack) can be further divided into a symbolic jump and a fixed jump. The former can jump into two possible states with the different symbolic values, and instead, the latter can only jump to a specific state according to the condition value, *i.e.,* True or False. As Fig. 15 shows, the conditional jump of Block #143 belongs to the symbolic type, because its condition value

[6]The specific operations of each opcode are detailed in [17].

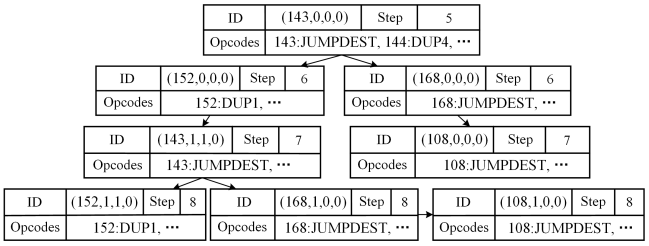


Figure 16: Example of State Exploration Tree based on CFG.

is determined by the parameter of contract calls, which are replaced by symbol values during the state exploration process. In other words, the symbolic value directly determines the end of the loop, that is, executing the loop body (Block #152) or the next step (Block #168) at that time. In this way, the exploration may stick in "infinite" loops and invalidate the most symbolic execution-based methods such as Manticore, since the condition value can always be True until the symbolic parameter "x" reaches its maximum, *i.e.,* 2^{256}. To address this problem, a state naming mechanism and a loop detection algorithm are proposed to automatically identify loop operations and prune the "repetitive" jump destinations during the state exploration. In particular, the limited loop can be executed directly when the symbolic value is replaced by a fixed call value during the contract behavior analysis, since the conditional jump has a specific loop number.

State naming mechanism. In order to record and distinguish the executed states, while storing useful information for loop detection and priority search, a naming mechanism is designed for states. As depicted in Figure 16, an State Exploration Tree is build on the CFG example shown in Figure 15. Each of nodes holds an unique ID (*i.e.,* name), as well as the step and opcode. Among them, ID is represented in Eq. 12, and $ID_{x_1} = (0, 0, 0, Seq_{Tx})$ for the initial state x_1.

$$ID_{State} = (Startop_{State}, Seq_{State}, Num_{State}, Seq_{Tx}) \tag{12}$$

Specifically, (i) $Startop_{State}$ is the pc of the start opcode, which indicates the serial number of CFG block. (ii) Seq_{State} is the serial number of the block, which can distinguish the same block in different states, especially for the loop paths. For example, as shown in Figure 16, the Block #168 can be executed in multiple paths, such as [#143, #168] and [#143, #152, #143, #168]. To distinguish them, according to Eq. 13, the numbers 0 and 1 are set for them, respectively.

$$Seq_{x_{t+1}} = \begin{cases} Num_{x_t} + 1 \text{ if } Num_{x_t} > 0 \\ Seq_{x_t} \quad \text{otherwise.} \end{cases}, Seq_{x_1} = 0, t \geq 1 \tag{13}$$

(iii) Num_{State} refers to the repeat number of the block and can be calculated by Eq. 14. Notably, it is designed to facilitate the judgment of loop states, as shown in line 8 of Algorithm 1.

$$Num_{x_{t+1}} = \sum\nolimits_{i \in [1,t]} 1 \left[Startop_{x_i} == Startop_{x_{t+1}} \right] \tag{14}$$

where $1[\cdot]$ is 1 only when the condition $\cdot$ is true, and $Num_{x_1} = 0$. (iv) Seq_{Tx} reveals the transaction sequence number, and it is critical to label the specific order of vulnerable paths in their cooperation sequence. In this way, it can enable the machine learning-based agent to guide the symbolic executor to detect vulnerabilities that require multiple execution paths (*i.e.,* transactions), and some details are illustrated in § C.4. In total, the first three items can distinguish the states in a path, and all of items can specific a state in any execution step.

Loop path discovery. Thanks to the designs of naming mechanism, SymX can employ the item Num_{State} to judge the loop states quickly, *i.e.,* $Num_{State} > 0$. Furthermore, in order to explore the path completely and efficiently, it introduce a threshold $R = 2$ to unroll the loop body, which is the same as current methods [62]. That is, as illustrated in lines 8∼9 of Algorithm 1, the loop state x_t is limited when its $Num_{State} \geq R$. Nevertheless, breaking loop states directly is inappropriate for some situations. For example, for the statement "for(int i=0;i<5;i++)", states with clear cycle numbers will automatically end the loop on time and they have a specific next state for each step. To this end, SymX select unexplored next states of the loop states only when they have two possible next states, *i.e.,* their conditions are symbolic and both can be satisfied by calling contracts with different parameters. In this way, SymX can address the "infinite" loop problem mentioned in Figure 15 during

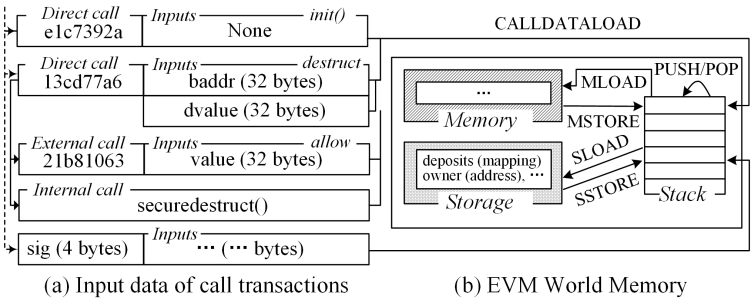


Figure 17: Examples of call transaction sequences and EVM World Memory.

the vulnerability detection. Also, the process of behavior analysis does not trigger the loop limitation given that they have specific execution objects, so that their execution processes are complete and the result correctness can be ensured.

C.3 Symbolic Constraint Solve

Constraint Solvers, as the basic component of the symbolic execution technology, are responsible for validating the feasibility of states based on Satisfiability Modulo Theories (SMT) during the process of state exploration, so as to ensure paths can be attached, which is illustrated in line 7 of Algorithm 1. That is, the next states can be executed when Constraint Solvers output the satisfiable (SAT) conclusion according to the current symbolic constraints. This can eliminate some false positives that cannot be executed successfully, *i.e.,* vulnerable dead code of contracts [9]. Also, they are employed to solve the call parameters that can meet the accumulating constraints of vulnerable paths, which can be further used to simulate/validate attacks and make auditors understand vulnerabilities better, *c.f.,* § C.7. Notably, thanks to the translation mechanism of SMT Library (SMT-LIB) languages [51], SymX can invoke multiple SMT solvers [3] by adding their interface easily, such as Z3 [14], Yices [16], and CVC4 [2]. Users can use either of them or their combination [54], given their unique performance and built-in theories. For instance, Z3 can handle linear/non-linear operations on more data types such as bitvectors and arrays, while Yices can achieve a faster solving process [24]. Furthermore, on this basis, SymX employs iterative constraint solving to obtain the n different solutions at a time, so that diverse vulnerable inputs can be obtained quickly and help auditors analyze vulnerabilities more clearly.

C.4 Transaction Sequences and EVM World Memory

Launching a contract attack may require the cooperation of multiple internal or cross calls, which are dedicated to satisfying the attacking conditions. For instance, as depicted in Figure 17, two direct calls are conducted by an SU attacker, and the second one will first generate a cross-contract (also named external) call transaction "b.allow(dvalue)" to get the condition of the "require" operation, *i.e.,* true or false. However, most of symbolic executors, such as SmartTest, directly use *false* rather than actually execution to reduce the false reports caused by external calls. Even more, the unexpected behaviors of fallback function "function()", *e.g.,* recalling the "init()" function to update the "owner", cannot be considered when the *Booloracle* contract receives the Ether. This will cause incorrect results, such as the wrong owner. In contrast, SymX can symbolically execute the external function "allow(value)" by initiating an inline call transaction with function id of "allow(uint256)" and symbol "value" in the World Memory to explore the execution paths of *Booloracle* contract. Also, this enable the path checking for fallback function when it is triggered.

Furthermore, the internal call "securedestruct()" will continue to be conducted by the second direct call, so that its state execution path can be constructed completely. Note that, the statement "require(msg.sender==owner)" defines that the state variable "owner" is the key factor for the successful execution and can be updated by the first direct call "init()". Thus, in order to maintain these state variables within multiple transactions, SymX records them into the *Storage* of the World Memory timely, which holds the global state of the contract and spans the life across different executions. Specifically, as depicted in Figure 17, the call parameters (symbolic and concretized values in vulnerability detection and behavior analysis, respectively) are first loaded into the EVM Stack by

Symbolic Executor through CALLDATALOAD instructions during the call execution, and then the opcodes of the called functions are executed based on the EVM Stack (PUSH/POP), Memory (MLOAD/MSTORE), and Storage (SLOAD/SSTORE). Among them, the EVM Stack manages the symbolic or concretized values to guarantee the opcode execution, as illustrated in § C.2. The Memory is used to temporarily store data chunks whose size usually exceeds 256 bits (the data size of a stack element). The Storage is organized as a key-value table [50]. Simple fields can be directly accessed with their compiler-determined indices. Complex fields like the mapping balances require hash values as the keys. For example, the value of "deposits[msg.sender]" can be retrieved by visiting the slot corresponding to the hash value of "msg.sender$\odot indice_{\text{deposits}}$", where the symbol $\odot$ denotes a concatenation operation.

Besides, the EVM specifications [20] are strictly followed in the World Memory, such as execution state rollbacks and Gas costs for executing instructions, so as to execute the contract correctly. Thus, these designs are essential for the detection of transaction cooperation and transaction-related vulnerabilities such as TOD, as well as the behavior analysis of multiple transactions.

C.5 Vulnerability Detection

As shown in Fig. 2, SymX employs multiple Vulnerability Detectors listed in Table 1 to track the state changes and identify the defective opcode fragments during the path exploration. Notably, among the vulnerability detection, the taint analysis technology is used to judge the connection of variables conveniently along the path execution, which also can make the detection logic more robustness and resist code obfuscation. Meanwhile, the transaction states can be introduced in detection of some vulnerabilities such as RE and UCL to verify whether inline calls is executed successfully. Furthermore, RE, IO, AS, and TOD are required to be detected in the paths that end normally, *i.e.,* the end state is not listed in [REVERT, THROW, ERROR], since they can only make impacts in this way and this is a main reason for false positives of other tools [51, 49]. Specifically, the detection logic of some important vulnerabilities are detailed as follows.[7]

reentrancy-eth (RE). Ethereum provides three methods to transfer Ethers, *i.e.,* address.send(), address.transfer(), and address.call().value(). These methods all generate a CALL instruction, which reads seven values from the EVM stack. The first three values represent the gas limitation, recipient address, and transfer amounts, respectively. The CALL instruction that meets the following conditions is almost generated by call().value(). (i) The gas limitation is larger than "2300"; (ii) the transfer amount is greater than 0. In addition, the SLOAD instruction is used to get a offset value (named *Slot ID*) from the EVM stack and puts the result read from storage back onto the stack [75]. If an SLOAD instruction is executed before the CALL instruction and its *Slot ID* are written by an SSTORE instruction after executing the CALL instruction, the CALL instruction can be executed again and cause Reentrancy problem.

suicidal (SU). Unprotected calls to a function executing selfdestruct or suicide statements, which generate a SELFDESTRUCT instruction to destroy the contract. This may cause property damage and business interruption. In order to describe the unprotected behavior, the arbitrary caller address can be defined as there is at least one possible value of the symbolic caller, or the caller is "msg.sender". In this way, this vulnerability can be detected by verifying that (i) the SELFDESTRUCT instruction can be executed and (ii) the caller belongs to the arbitrary addresses.

arbitrary-send (AS). This vulnerability refers to the Ether transfer operation can be executed by everyone, since the callers are not checked. In this way, it can be detected when the following three conditions are satisfied simultaneously. (i) The CALL instruction of the Ether transfer operation is reachable. (ii) The caller is a arbitrary address. (iii) The Ether value is larger than zero.

tod-ether/receiver (TOD). Different transaction execution orders may result in distinct results, so that attackers can prioritize their transactions and manipulate results by setting higher Gas. Thus, this vulnerability is detected by checking two symbolic transactions that meet the following conditions. (i) They read/story the same global state variables, *i.e.,* the same offset of SLOAD/SSTORE instruction. (ii) The different results can be obtained by swapping their sequences.

integer-overflow (IO). The overflow of arithmetic operations of signed and unsigned integers, including addition, subtraction, multiplication, division, modulo, and negation, can result in unexpected

[7] More bugs in Table 1 are detailed on `https://github.com/Secbrain/RSymX/tree/main/Detectors`.

results such as withdrawing more Ether than previously deposited. In order to detect the possible overflow operations, the calculation results of instructions such as ADD/SUB/MUL is taint with the label "IOS/IOU" (sighed/unsighed integers), and then check them when the computation is complete by using solvers such as Z3 to solve the inputs that can trigger the integer overflow. Among them, the range of value type (*e.g.,* Uint8) is the key to check values whether out of their scope, while the current tools such as Manticore [51] only works on Uint256/int256. Also, the paths that exist the above situation but end abnormally may be false positives because the exceptions are caused by a "require" or "assert" statement that have verified the overflow.

C.6 Abnormal Contract Behaviour Identification

In addition to complete the vulnerability detection and obtain the exploit call parameters, the security of contract call transactions collected by Transaction Inputer can also be analyzed by validating their executed state paths directly. Specifically, as depicted in Figure 2, Symbol Executor can first extract the invoked functions and parameters from contract call transactions. Their examples are shown in Figure 17(a). The function id is a four-byte signature, and the parameters are a series of 32-byte data arranged in order according to the function API. If there is a variable-length array, *e.g.,* "uint256[]", the corresponding slot saves the offset to its actual contents, including the length and specific elements. Meanwhile, different from the fixed-length variables, it will be loaded to the Memory through CALLDATACOPY instructions and then stored in the EVM Stack for further operations. Besides, contract call transactions also include transaction hash, block number, timestamp, from/to (caller/contract) address, call value, Ether price, *etc*. Note that all information will be used for security analysis, since they can be loaded by the corresponding opcodes, such as CALLER, CALLVALUE and TIMESTAMP.

Second, Symbol Executor replaces the symbolic values in the above state exploration process with fixed parameters to obtain the specific state execution paths based on the contract code (either of source code, bytecode, and opcode). Note that, the code can be replaced with the contract deployed transaction, including the initial bytecode and arguments, and Symbol Executor can execute it to deploy the contract, which also can restore the contract status losslessly. Then, similar to vulnerability detection, multiple Vulnerability Detectors are employed to check the security of the explored paths, and the detection results are sent to Result Exporter for output. Notably, thanks to the component of EVM World Memory, the abnormal behaviors of cross-transactions and cross-contracts can be identified accurately, whose principle is the same as the detection process of multiple transaction collaboration vulnerabilities, as detailed in § C.4. For instance, two safety transactions related to the TOD vulnerability and performed in a block will lead to inconsistent results given their different execution orders, and this transaction sequence can be accurately identified as TOD abnormal behaviors by SymX.

C.7 Vulnerable Instance Validation

Through the above vulnerability detection and transaction analysis process, the defects of contracts can be discovered to standard the source code development. Nevertheless, some of vulnerabilities identified by the pre-mentioned Vulnerability Detectors may not be exploited by attackers to steal extra amounts. To this end, the refinement of vulnerability detection results is useful to find the genuinely harmful discoveries, which should be fixed by the developers as soon as possible. Specifically, Vulnerability Validator works automatically based on the following two sub-components.

The validation templates for vulnerabilities. The first step of verification process is to deploying the vulnerable contracts and replaying the potential attack calls. The deployment requirements is the same as behavior identification (*c.f.,* § C.6), that is, either contract code or deployment transactions. Next, the methods of initialing calls are divided into two types according to their vulnerability exploitation.

(i) Exploit call replaying. All of the potential calls are directly replayed in their order based on the EVM World Memory for vulnerabilities that can be triggered simply, such as AS and SU. For example, after the execution of AS calls, the last call is designed to be performed once additionally, so that the functions that can be invoked without restrictions can be discovered.

(ii) Auxiliary contract executing. There are some vulnerabilities that need to be exploited through auxiliary contracts, such as RE. For them, SymX develops the auxiliary contract templates to make calls indirectly. As an example shown in Listing. 3, the *ReentranceExploit* contract is designed

for the RE vulnerability. Vulnerability Validator deploys this contract and sets the address of the vulnerable contract by invoking the "set_vulnerable_contract" function. Then, he replays each call data through "proxycall" function. During the call process, the fallback function "function()" is triggered when the auxiliary contract receives the Ether, so that the call data can be conducted repeatedly until "reentry_num=0". Finally, he destroys this contract and obtains its balance by invoking the "get_money" function. In this way, the vulnerable contract suffers from a RE attack.

```
contract ReentranceExploit {
  uint reentry_num = 2, reentry_val = 0;//Reentry values
  address owner, vul_contract; //Attack settings
  bytes reentry_str; //Reentry data
  function ReentranceExploit() payable{
    owner = msg.sender;
  }
  function set_vulnerable_contract(address _contract){
    vul_contract = _contract;
  }
  function proxycall(bytes data, uint val) payable{
    reentry_str = _reentry; reentry_val = val;
    vul_contract.call.value(val)(data); //Initial calls.
  }
  function get_money(){
    suicide(owner);//Destroy contract and get assets.
  }
  function() payable{
    if (reentry_num > 0){
      reentry_num = reentry_num - 1;
      vul_contract.call.value(reentry_val)(reentry_str);
    }
  }
}
```

Listing 3: Example of auxiliary contract templates for launching attacks.

The goals/metrics to measure the success of attacks. After replaying the vulnerable calls, SymX determines the attacks are successful when the goals/metrics are achieved, currently including the following four types.

(i) Balance status tracking. The economic gain is a fundamental factor that needs to be considered, since most vulnerabilities, such as RE and AS, are exploited by attackers to increase their balance. Thus, after a series of attack calls, the account balance of attackers increases, indicating that the vulnerability exploit is successful. Otherwise, the vulnerability is not as dangerous as expected and can be considered a false positive.

(ii) Contract/Transaction status monitoring. In addition to stealing the contract balance, some vulnerabilities aim to break contracts or their logic. To describe their influence, other EVM world attributes (*e.g.,* the status of contracts and transactions) are monitored to verify the correctness of the results. For instance, after the SU calls, the contract was destroyed, *i.e.,* its bytecode is reset to "0x" in the EVM world. Meanwhile, the balance of attackers will be increased if the contract balance is non-zero. Besides, the return values of TOD transactions differ at varying execution orders, and Gas exhaustion occurs during calls with DOS vulnerability.

(iii) Variable status tracking. There are some vulnerabilities whose ultimate impact is difficult to measure, such as IO and UIO. In order to validate them, Vulnerability Validator tracks the variables and checks their operations in real-time based on the EVM Stack. For instance, abnormal computation results occur in the state execution paths of calls with IO vulnerability. The value of global state variables are changed incorrectly during the execution of calls with UIO vulnerability.

(iv) Source code checking. Thanks to the mechanism of defective source code mapping, code checking can be used to validate defects that reflect the significant code features. For instance, the *revert-require* optimization can be directly verified by checking the corresponding defective source code is "revert()" rather than "require()". The UIS defect can be validated by checking the state variable whether is initialed with a specific value. Notably, there are some defects do not require validating with dynamic execution, such as LLC, BE, and ST. It can be attributed to the facts that they can be validated simply by checking the contract source code.

C.8 Result Exporter

After the vulnerability detection, behavior identification, or their result validation, the Result Exporter generates a security analysis report for the contract, which consists of security conclusions and repair suggestions.[8] Specifically, among the vulnerability detection, he outputs the "safety" conclusion and corresponding evidence (*i.e.,* analysis details) when there is no vulnerable paths. Otherwise, the contract vulnerabilities with severity (*i.e., High*, *Medium*, *Low*, *Info*, and *Opt*) are reported. Meanwhile, the defective source codes are proposed by mapping vulnerable opcode fragments. Meanwhile, he can output exploit call parameters and possible repair suggestions to enable developers to simulate, understand, and fix vulnerabilities better. Additionally, if the users perform the transaction identification and vulnerability validation, the corresponding results will be output in the final reports.

Locating the defective source code statements. After the vulnerability detection, Result Exporter obtains the vulnerability types and their defective opcodes $[(vul, ops_{vul})]_{vul \in \mathcal{V}}$ from Vulnerability Detectors $\mathcal{V}$. As represented in Eq. (15), when the source code of the contract c is provided, he can map the begin and end positions (*i.e.,* lines and columns) of these opcodes in the source codes based on the ASM file extracted by Contract Inputter. Then, the defective contract statements $c_{\mathcal{V}}$ are obtained by intercepting the source codes of the specific positions.

$$c_{\mathcal{V}} = \Big[\Big(vul, [c\,[begin{:}end \leftarrow \mathrm{ASM}(op)]]_{op \in ops_{vul}}\Big)\Big]_{vul \in \mathcal{V}} \tag{15}$$

Secondary utilization and development. In order to help developers understand contract vulnerabilities more intuitively, SymX also provides an API interface to output contract CFG figures and indicate the vulnerable state paths, so that they can clearly learn the process of vulnerabilities being triggered. Also, SymX is not only a contract security tool, but also an educator in safety code development based on contract specifications [18].

D Supplementary of State Features

Table 3 lists d unique features extracted for a symbolic state based on its information, such as ID, block instructions, and execution depth. Specifically, the features consist of ten types. The first five types require information on the true and false branches. (1) The *covnumNew* item refers to the number of new instructions achieved by the state and its branches, given that the states with more new coverage should be explored first. (2) On this basis, the *covratioNew* item calculates the coverage ratio of new instructions to mitigate differences of varying contract sizes. Furthermore, to distinguish the repeated states, (3) the number of their instructions *covnum* and (4) the corresponding coverage ratio *covratio* are counted. (5) Also, the visited times of states *visitTimes* is used to identify repeated states, *i.e.,* the value of Num_{State} in their ID.

The last five features are only calculated on the current states. (6) The *constraint* item refers to the size of constraints at the end of the state, which can reflect the complexity of the current state path. (7) Similarly, the size of the last stack *stack* is calculated since the larger the stack, the deeper the execution. (8) The total execution time of instructions *time* is monitored to make the model perceive the efficiency of the execution, which is one of the goals of RSymX. (9) The transaction serial number *transeq* (*i.e.,* Seq_{Tx} in ID) is recorded to indicate the states in different execution, because most vulnerabilities rely on the execution of multiple transactions. Besides, inspired by existing expert-designed heuristics [7, 40], (10) the execution depth of state *depth* in the current transaction is counted to further specify the state and enable the model to learn the advantages of those heuristics. In total, based on these features, 3DQN can describe states clearly. Given its decoupled implementation, more features can be supplemented to support large language models in the future.

E Supplementary of Strategy Learning

This section provides the detailed procedure of the learning-guided state exploration used by RSymX. Algorithm 1 describes how the symbolic executor performs dynamic depth-first search (DDFS) with the learned scheduler. For each contract and transaction round, RSymX extracts state features, obtains

[8]The examples of audit reports are shown in `https://github.com/Secbrain/RSymX/tree/main/Reports_examples`.

Algorithm 1 State exploration with learning strategies

Input: Contracts C; state space mapping function of contracts $\xi : c \in C \rightarrow X$; feature extraction function of states $\phi : x \in X \rightarrow s$; transaction number threshold TX; loop number threshold R; probability threshold ε; number threshold of explored paths n_{paths}.

Output: The vulnerable state paths and trained 3DQN model, including policy and target networks.

Initialize experience replay/temp buffers $D = D_r = \emptyset$.
Initialize the policy/target networks Q/$\hat{Q}$ with random parameters W_p and $W_t = W_p$.
Initialize vulnerable paths $vpaths = \emptyset$ and the predicting numbers of policy network $n_Q = 0$.
procedure DDFS(x_t, s_t, $path$, c) ▷ Dynamic DFS.
  if $\|paths\| \geq n_{paths}$ **then** ▷ Exceed the limitation.
    return ▷ End the exploration against path explosion.
  $ID_{x_t} = (Start_t, Sep_t, Num_t, Seq_{tx})$ refers to Eq. (12).
  Collect reachable states X_{t+1} by Symbolic Executor/Constraint Solvers, and $path \cup = x_t \in X_t$.
  if $Num_t \geq R$ && $\|X_{t+1}\|$ == 2 **then** ▷ Loop branch.
    $X_{t+1} = [x_{t+1} \,|Num_{t+1} == 0, x_{t+1} \in X_{t+1}]$.
  if $\|X_{t+1}\|$ == 0 **then** ▷ End of the path, and check it.
    Record defects $vpaths \cup =$ ($path$, $vuls$, $inputs$, Seq_{tx}, c) of $path$ by Vulnerability Detectors.
    Store the explored paths $paths \cup =$ $path$.
  else if $\|X_{t+1}\|$ == 1 **then** ▷ Only one next state.
    DDFS($X_{t+1}[0]$, $\phi(X_{t+1}[0])$, $path$, c)
  else ▷ Two next state caused by symbolic condition.
    Obtain the action sequence A_t with ε by Eq. (9).
    Extract the best action $a_t = A_t[0]$, next state $x_{t+1} = X_{t+1}[a_t]$, and its features $s_{t+1} = \phi(x_{t+1})$.
    Store the state transform data into temp buffer $D_r \leftarrow D_r \cup (s_t, a_t, s_{t+1})$, and n_Q += 1.
    for all action $a_t \in A_t$ **do**
      DDFS($X_{t+1}[a_t]$, $\phi(X_{t+1}[a_t])$, $path$, c)
  Q,$\hat{Q}$,$paths$,D,D_r = Update_3DQN(Q,$\hat{Q}$,$paths$,$vpaths$, D,D_r,n_Q) ▷ Refer to Algorithm 2.
(Main process) *Traverse contracts and check them.*
for all Contract $c \in C$ **do**
  for all Transaction $Seq_{tx} \in [1, TX]$ **do**
    Initialize state $x_1 \in X_1 \in X = \xi(c)$ and $\|X_1\|$ == 1.
    Initialize explored paths $paths = \emptyset$, and $s_1 = \phi(x_1)$.
    Call DDFS(x_1, s_1, $\emptyset$, c) to get the vulnerable paths.
return $vpaths$ and trained 3DQN model (Q,$\hat{Q}$)

candidate successor states from the symbolic executor and constraint solver, and applies loop/path limits to control state explosion. When a symbolic branch has two successors, the 3DQN agent ranks the exploration order using the learned policy and records the transition into a temporary buffer. The discovered vulnerable paths and explored states are then used to update the agent online.

Algorithm 2 details this online update process. After enough paths are collected, RSymX builds the test tree to compute coverage rewards, combines them with vulnerability rewards, and transfers transitions from the temporary buffer to the experience replay buffer. The policy network is updated periodically using sampled experiences, while the target network is synchronized at a slower interval to stabilize learning.

F The Supplementary of Experiment Settings

For Dataset$_1$, the ground truth consists of labeled vulnerable items, each specified by a vulnerability type and its code location, while safe contracts are represented by empty vulnerability lists. We therefore evaluate tools in a fine-grained positive-item setting. A reported item is counted as a true positive (TP) if its vulnerability type and code location match a labeled vulnerable item. A false negative (FN) is a labeled vulnerable item that is not reported by the tool. A false positive (FP) is a reported vulnerable item that does not match any labeled vulnerable item under the available labels. Because tools only report vulnerable locations rather than an explicit set of safe code lines,

Algorithm 2 Update_3DQN

Input: The policy and target networks Q and $\hat{Q}$ (3DQN model); contract state paths $paths$; vulnerable paths $vpaths$; experience replay/temp buffers D/D_r; prediction numbers of policy network n_Q; update cycles of policy/target network C_p/C_t; path number threshold C_r

Output: The updated 3DQN model and $paths$, as well as buffers D and D_r

if $\|paths\| \geq C_r$ **then**
  Construct test tree with $paths$ to get rewards R_{cov}.
  Obtain rewards R_{vul} according to $vpaths$.
  for all $s_t \in D_r$ **do**
    Calculate reward $R_{\text{mix}}(x_t)$ of x_t. ▷ Refers to Eq. (5).
    $D \leftarrow D \cup (s_t, a_t, R_{\text{mix}}(x_t), s_{t+1})$.
  Clear temp buffer $D_r = \emptyset$ and $paths = \emptyset$.
if $n_Q \% C_p == 0$ && $\|D\| \geq n$ **then** ▷ Update policy network Q.
  Sample n records $[(s_t, a_t, R_{\text{mix}}(x_t), s_{t+1}) \in D]_{i \in [1,n]}$.
  Initialize the training loss $loss = 0$.
  for all $(s_t, a_t, R_{\text{mix}}(x_t), s_{t+1})$ **do**
    if x_{t+1} unexplored $\|X_{t+2}\| < 2$ **then** $y_t = R_{\text{mix}}(x_t)$.
    else Calculate y_t of state x_t in line with Eq. (10).
    $loss \leftarrow loss + (y_t - Q(s_t, a_t; W_p, B_p))^2/n$.
  Update parameters (W_p, B_p) of Q by gradient descent. ▷ Refers to Eq. (11).
if $n_Q \% C_t == 0$ **then** ▷ Update target network $\hat{Q}$.
  Update parameters $(W_t \leftarrow W_p, B_t \leftarrow B_p)$ of $\hat{Q}$.
return Q, $\hat{Q}$, $paths$, D, D_r ▷ Update to Algorithm 1.

Table 9: The kinds of vulnerabilities listed in Table 1 and detected by the methods.

Methods	High	Medium	Low	Info	Opt
SmartCheck [67]	-	IE,UCL,TO	TS	LLC,MEZ	ST
Slither [21]	RE,SU,CDC, AS,UIS,UIO	IE,UCL, TO,UCS	TS	LLC	BE
Park [83]	RE,SU,CDC, AS,TOD	IO,TO, UCL,UCS	TS,BP	-	-
Manticore [51]	RE,SU,CDC,AS, TOD,UIS,UIO	IE,IO, UCL,UCS	TS,BP	-	-
MythrilQL [72]	RE,SU,CDC, AS,TOD	IO,TO, UCL,UCS	TS,BP	-	-
DefectChecker [9]	RE	IE,UCL,TO,UCS	BP	-	-
SmartTest [62]	SU,AS	IO,TO	-	-	-
Oyente [49]	RE,TOD	IO	TS	-	-
ILF [31]	RE,SU,CDC,AS	UCL,UCS	TS,BP	-	-
RLF [66]	RE,SU,CDC,AS	UCL,UCS	TS,BP	-	-
SMARTIAN [13]	RE,SU,CDC,AS	IO,UCL,TO,UCS	TS,BP	-	-
TMP [84] SymX **RSymX (Ours)**	RE,SU,CDC, AS,TOD, UIS,UIO	IE,IO,UCL, TO,UCS	TS,BP	LLC,MEZ	ST,BE

and exhaustive line-level negatives are unavailable, true negatives (TN) are not calculated in a tool-independent way for this fine-grained setting, which is consistent with [67, 21, 49, 41]. Thus, the overlap score (OS), precision (P), recall (R), and F-Measure (F1) are calculated as follows to evaluate each method, where $\#TP$, $\#FP$, and $\#FN$ refer to the number of items marked accordingly.

$$OS = \frac{\#TP}{\#TP + \#FP + \#FN}$$

$$P = \frac{\#TP}{\#TP + \#FP}, R = \frac{\#TP}{\#TP + \#FN}, F1 = \frac{2 \times P \times R}{P + R}$$

By contrast, Dataset$_2$ is evaluated as a binary classification task (*i.e.,* normal and vulnerable), because its labels are defined at the contract level. In this setting, a true negative (TN) is a contract correctly

Table 10: Details of datasets. Among them, #L and #F represent the number of issues at the line and file levels. LINES, LENGTH, SIZE, and NUMS denote the total number of code lines, string length, byte size, and files.

DATASET	NUMBER OF ISSUES					CONTRACT DETAILS			
	HIGH	MEDIUM	LOW	INFO	OPT	LINES	LENGTH	SIZE	NUMS
$Dataset_1$ [15, 41]	$708^{\#L}$	$168^{\#L}$	$64^{\#L}$	$90^{\#L}$	$101^{\#L}$	15,807	236,355	448KB	158
$Dataset_2$ [9]	$12^{\#F}$	$32^{\#F}$	$42^{\#F}$	-	-	-	-	15.0MB	579
$Dataset_3$ [41]	-	-	-	-	-	7,954,979	276,378,424	284.3MB	13,413
$Dataset_4$ [42]	-	-	-	-	-	-	-	1.295GB	183,710
$Dataset_5$ [41]	-	-	-	-	-	7,565	264,514	265.0KB	27

Table 11: P and R results of vulnerabilities supported by each method on $Dataset_1$.

Severity	Metric	SymX	RSymX	SmartCheck	Slither	Park	Manticore	MythrilQL	DefectChecker	SmartTest	Oyente	ILF	RLF	SMARTIAN	TMP
High	P	96.64	**97.91**	-	87.81	92.93	90.74	90.36	95.00	89.88	83.88	95.39	96.12	92.28	81.84
	R	93.81	**95.68**	-	93.79	88.54	87.99	88.57	86.36	92.12	75.99	84.72	90.64	78.15	83.41
Medium	P	95.39	**95.77**	96.47	92.62	91.97	85.35	90.21	92.23	94.28	84.31	84.52	88.64	87.03	77.30
	R	94.01	**94.95**	86.27	91.33	83.73	83.01	80.07	90.17	91.80	75.44	78.63	88.30	80.98	84.93
Low	P	100.00	**100.00**	100.00	95.00	90.68	100.00	90.16	94.87	-	88.89	98.65	98.68	100.00	78.46
	R	90.59	**95.29**	61.90	90.48	80.01	90.59	75.30	86.05	-	76.19	84.72	88.26	90.59	89.48
Info	P	100.00	**100.00**	100.00	97.40	-	-	-	-	-	-	-	-	-	84.64
	R	89.29	**89.88**	66.07	89.29	-	-	-	-	-	-	-	-	-	77.98
Opt	P	95.52	**96.13**	100.00	95.18	-	-	-	-	-	-	-	-	-	82.23
	R	93.15	**93.77**	60.00	97.53	-	-	-	-	-	-	-	-	-	91.30
Overall	P	96.91	**97.58**	98.49	91.15	92.18	90.51	90.27	93.13	91.64	85.24	93.49	94.89	91.73	80.56
	R	92.93	**94.58**	73.27	92.72	85.24	86.85	83.06	88.85	91.99	75.90	83.20	89.46	81.77	84.78

classified as normal (*i.e.,* no vulnerability was detected), which allows us to compute standard accuracy (ACC). Among it, # means the number of contracts marked accordingly.

$$ACC = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN}$$

Additionally, Time (T) and Memory (M) overheads are evaluated for method efficiency.

G Supplementary Experiment Results

G.1 Discoveries of Wild Contracts

Table 12 reports the number of vulnerabilities detected by each tool on $Dataset_3$. Overall, RSymX finds 35,761 issues, while SymX reports 26,078, yielding about 37% more bugs under the same detection rules and only 0.68% analysis failures for both. Among them, about half were TOD vulnerabilities because they detected read-write conflicts between any two functions, which are difficult to exploit but risky. In particular, RSymX discovers substantially more other high–severity vulnerabilities such as RE, and also exposes more medium– and low–severity issues (*e.g.,* IO, UCL/UCS, and TS/BP). Traditional static pattern checkers (SmartCheck and Slither) are limited by their supporting vulnerability types, while classical symbolic engines (MythrilQL, DefectChecker, and SMARTIAN) either miss a large portion of bugs or suffer from much higher failure rates (up to 10.35% for DefectChecker and 99.86% for SMARTIAN). These results indicate that our execution semantics and vulnerability rules are expressive enough to cover diverse patterns, and that the dynamic strategy can drive the executor to more bug-revealing paths in production contracts.

```
1  contract VokenPublicSale2 is Ownable,... {
2    uint256 private _cacheWeiShareholders;
3    uint256 private _weiShareholders;
4    ...
5    function () external payable whenNotPaused {
6      uint256 __weiMax = _limit(msg.value);
7      if (__weiMax < msg.value) {
8        __usdAmount = _wei2usd(__weiMax);
9      }                                  Indirectly retriggered
10     else {
11       __usdAmount = _wei2usd(msg.value);
12     }                                  Duplicate
13     __usdRemain = __usdAmount;         withdrawals
14     ...
15     __usdUsed = __usdAmount.sub(__usdRemain);
16     __weiUsed = _usd2wei(__usdUsed);
17     if (__weiUsed > 0) {
18       ...
19       _weiShareholders = _weiShareholders.add(
           _cacheWeiShareholders);
20       (bool __bool,) = address(_SHAREHOLDERS).call.
           value(_cacheWeiShareholders)("");
21       assert(__bool);
22       uint256 __weiTeam = _weiSold.sub(_weiRewarded).
           sub(_weiShareholders).sub(_weiPended)...;
23       _weiTeam = _weiTeam.add(__weiTeam);
24       _TEAM.transfer(__weiTeam);
25       _updateSeasonLimits();
26     }                                  Update too late
27   } ...
28 }
```

Listing 4: The RE bug of *VokenPublicSale2*.

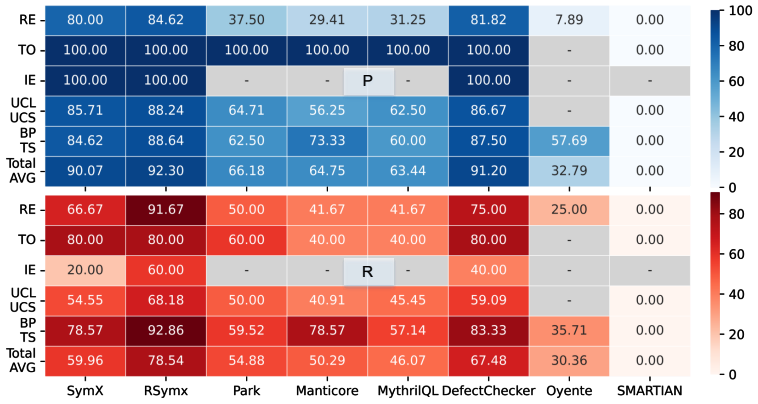


Figure 18: The average results of vulnerabilities supported by each method on Dataset$_2$.

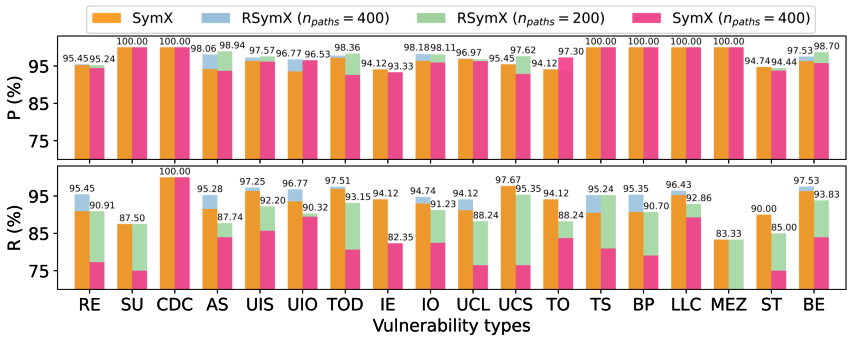


Figure 19: The detection results of RSymX with different exploration limits $n_{paths} \in [400, 200]$.

Detection of RE vulnerabilities. An RE vulnerability was discovered in the *VokenPublicSale2* contract (14 transactions, 2.9E+15Wei) by RSymX. As shown in line 20 (corresponding to line 1,060 of the code) of Listing 4, the call.value operation of the fallback function can be executed reentrantly, yet the remaining amount "__weiTeam" was too late to be transferred. In other words, when the *VokenPublicSale2* contract receives the Ethers, its fallback function will be executed, and the "_SHAREHOLDERS" can deliver a RE attack to withdraw more property belonging to "_TEAM". A similar vulnerable code is existed in the *VokenPublicSale2-1* contract (157 transactions). However, this vulnerability was missed by most methods such as SmartTest and MythrilQL. Thus, during the development stages, we should avoid the use of call.value and employ the transfer operation to withdraw funds.

Furthermore, we evaluate scalability and prevalence of vulnerabilities on 10,000 randomly sampled bytecode contracts in Dataset$_4$ from the public blockchain. Figure 21 visualizes, for each vulnerability type, the number of issues reported by different analyzers together with their global failure counts. RSymX again reports the largest number of potential problems, an improvement of about 42% for SymX, while keeping the failure rate at 0.68%, comparable to SymX and clearly lower than most baselines. This suggests that our approach can be used as a practical, large-scale scanner for deployed contracts.

Meanwhile, Figure 22 depicts all vulnerability discoveries of RSymX on Dataset$_4$ and correlates the detected vulnerabilities with the on-chain activity of contracts. We observe that vulnerable contracts span a wide range of transaction volumes. Many bugs appear in contracts with only a few interactions, but there also exist non-negligible numbers of high- and medium-severity vulnerabilities (*e.g.,* RE, IO, TOD) in contracts with thousands of transactions. This confirms that real financial value is increasingly under threat and highlights the need for accurate and scalable analyzers such as RSymX.

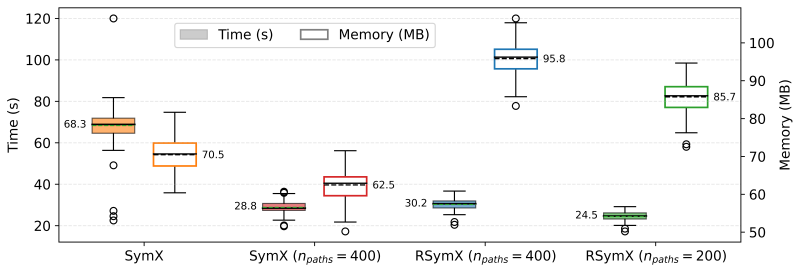


Figure 20: The time and memory overheads of RSymX with different exploration limits n_{paths}.

Table 12: Comparison of vulnerability number detected by tools on Dataset$_3$.

Severity	Project	SymX	RSymX	SmartCheck	Slither	Park	Manticore	MythrilQL	DefectChecker	SmartTest	Oyente	ILF	RLF	SMARTIAN	TMP
High	RE	434	651	-	295	70	426	46	362	524	12	317	495	0	1,294
	SU	80	96	-	77	42	56	31	-	61	-	47	54	16	3,156
	CDC	76	88	-	109	47	61	34	-	-	-	41	67	0	6,453
	AS	346	520	-	1,197	-	295	-	-	352	-	255	333	0	664
	TOD	13,119	17,279	-	-	-	8,904	-	-	-	234	-	-	-	3,531
	UIS	267	534	-	217	-	220	-	-	-	-	-	-	-	594
	UIO	127	203	-	10	-	92	-	-	-	-	-	-	-	424
	Total	14,449	19,371	-	1,905	159	10,054	111	362	937	246	660	949	16	16,116
Medium	IE	564	731	56	1,573	-	579	-	81	-	-	-	-	-	4,121
	IO	3,792	5,868	-	-	312	6,121	257	-	4,826	5,058	-	-	253	12,254
	UCL	591	739	214	153	69	375	56	1,257	-	-	304	430	220	5,485
	TO	36	44	383	117	30	26	22	279	39	-	-	-	19	4,551
	UCS	181	227	-	127	82	153	65	1,257	-	-	135	177	220	705
	Total	5,164	7,609	653	1,970	493	7,254	400	2,874	4,865	5,058	439	607	712	27,116
Low	TS	1020	1,457	26	3,147	-	872	-	-	-	73	752	917	1	2,784
	BP	708	945	-	-	242	518	207	680	-	-	471	1,123	1	2,043
	Total	1,728	2,402	26	3,147	242	1,390	207	680	-	73	1,223	2,040	2	4,827
Info	LLC	3154	4,017	1,696	4,454	-	-	-	-	-	-	-	-	-	2,956
	MEZ	212	265	246	-	-	-	-	-	-	-	-	-	-	894
	Total	3,366	4,282	1,942	4,454	-	-	-	-	-	-	-	-	-	3,850
Opt	ST	404	646	83	680	-	-	-	-	-	-	-	-	-	3,486
	BE	967	1,451	-	1,432	-	-	-	-	-	-	-	-	-	435
	Total	1,371	2,097	83	2,112	-	-	-	-	-	-	-	-	-	3,921
Overall	Total	26,078	35,761	2,704	13,588	894	18,698	718	3,916	5,802	5,377	2,322	3,596	730	32,652
	Failed	96	96	0	191	235	87	242	749	235	8,742	177	183	1,327	0
	Secure	1,839	797	10,915	544	12,702	4,995	12,781	8,935	7,514	1,361	4,252	3,141	11,580	97

H Supplementary of Discussion

H.1 Visualization of vulnerable paths

RSymX can render the CFG of contracts and highlights vulnerable paths to explain how exploits materialize, particularly useful for bytecode-only cases. Figure 24 visualizes the example of Listing 6. The *Suicidal* contract was identified with an SU vulnerability, which is triggered by the following steps.

1. The attacker invokes the function “init()” to set the variable “owner” with his address.
2. The attacker invokes the function “false_selfdestruct()” to bypass the “require” verification and destroy the *Suicidal* contract.

Blocks outlined in red frames form the second vulnerable path, which causes the SU vulnerability directly. Both SymX and RSymX confirm that the SELFDESTRUCT instruction at PC 242 is reachable by any caller after the above two calls. Notably, this contract holds 36 paths when there are two transactions. SymX explores them via repeated conditional jumps (*e.g.,* Blocks #13, #81, #65). In contrast, guided by the 3DQN model, RSymX prioritizes high-yield traces, *i.e.,* the paths [0,13,65,104,116,243,125] and [0,13,81,93,127,218] during two transactions, respectively.

Moreover, this case illustrates the effectiveness of a hybrid reward scheme. (i) coverage–only shaping rewards novelty at each step, so it aggressively grows long, previously unseen branches under blocks like #13/#65 and spends budget on benign detours. Although recall/coverage increases, rewards are dominated by path length and novelty, delaying the prerequisite action “init()”, which unlocks the sink. (ii) Bug–only shaping does the opposite. It concentrates reward near patterns correlated with high–severity sinks (*e.g.,* SELFDESTRUCT), but without the precondition satisfied, the path ends in

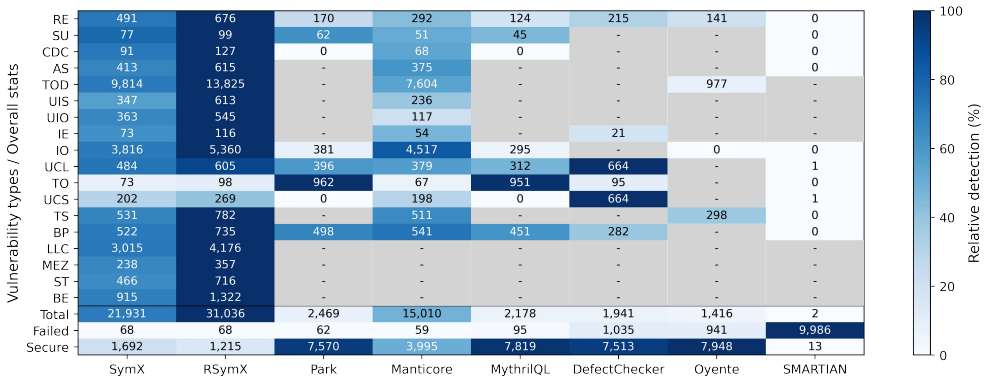


Figure 21: Numbers of potential issues and failures reported by tools on 10,000 bytecodes in Dataset$_4$.

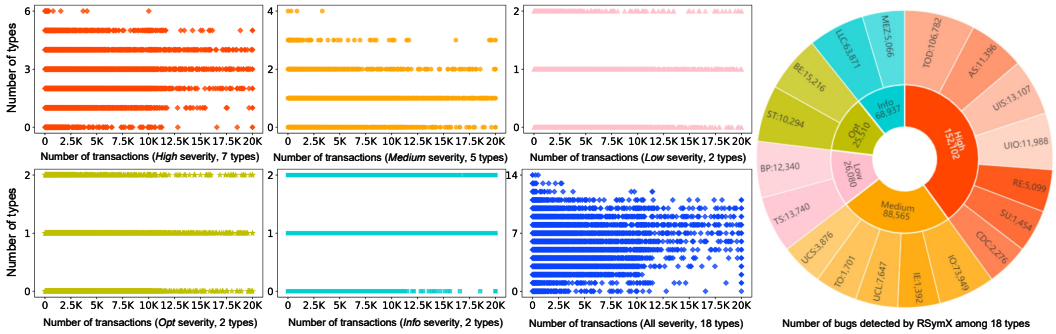


Figure 22: Correlation between contract transactions and vulnerability types detected by RSymX on all bytecodes in Dataset$_4$. In the figure, the upper limit of the number of transactions is set to 20,000.

a revert. Then, the agent repeatedly exploits similar high–risk neighborhoods (local optima), yielding poor coverage and missing the dependency within two calls.

To this end, our mixed reward reshapes the value function across the sequences: (i) time normalization discourages "orbiting" paths with low novelty per time and makes fast-to-novelty trajectories (*e.g.,* the first path) dominate in rewards, so the agent reaches the "false_selfdestruct()" region earlier over aimless exploration, while without sacrificing overall coverage (consistent with the coverage gains in Fig. 10). (ii) Severity weighting assigns a larger terminal payoff to high-severity events such as SELFDESTRUCT (the second path), and propagates delayed positive reward back to the earlier init() call. It makes "init–then–destroy" sequences strictly more valuable than either "explore for novelty" or "chase the sink" alone, yielding both higher precision and recall (matching the precision/coverage trade-offs in Fig. 10). Finally, they produce complementary guidance that consistently promotes the shortest, high-yield path satisfying cross-transaction dependencies to reach PC 242 with fewer explorations, whereas single-objective schemes either over-explore (more FP, delayed prerequisites) or over-exploit (poor coverage, more FN). This aligns with the quantitative trends in Fig. 10, where the mixed reward achieves the highest overall F1 while maintaining favorable runtime/coverage.

H.2 Feature-Aware Scheduling Analysis

To understand how RSymX makes decisions, we analyze the learned random-forest policy with global feature importance and local SHAP attributions. The policy maps the 20-D state vector in Table 3 to a binary branch-selection action.

Global importance. Figure 12(a) shows that $covnum_{\text{false}}$, $covnumNew_{\text{false}}$, *transeq*, and *stack* dominate decision making. These features summarize branch-local novelty, transaction context, and execution depth. By contrast, complexity indicators such as *constraint* and $covratio_{\text{current}}$ have smaller weights, suggesting that the policy does not rely on them in isolation. This ordering matches our design: the mixed reward mainly encourages time-normalized novelty and validated vulnerability discovery, making novelty-related features most predictive.

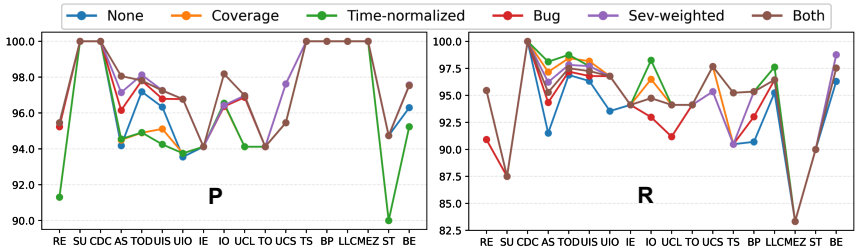


Figure 23: Performance of various reward schemes on Dataset$_1$ ($n_{\text{paths}=400}$, $TX=2$, $R=2$).

```
contract Arbitrarysend {
  mapping (address => uint) userBalance;
  function arbitrary_send() public {
    uint cc = userBalance[msg.sender];
    userBalance[msg.sender] = 0; //Clear balance
    msg.sender.send(cc); //False report
  }                          Only withdraw own balance
  function addToBalance() payable{
    userBalance[msg.sender] += msg.value;
  }
}
```

Listing 5: Example of AS vulnerability.

```
contract Suicidal {
  address owner;
  function init() public {  ①
    owner = msg.sender; //Not check it
  }

② function false_selfdestruct() public{
    require(msg.sender == owner);
    selfdestruct(msg.sender);
  }
}
```

Listing 6: Example of SU vulnerability.

Local explanations. Figures 12(b)-(c) show two representative states. At PC 64 with $Seq_{Tx} = 0$, low values of $covnum_{\text{false}}$ and $covnumNew_{\text{false}}$ produce negative SHAP contributions, pushing the prediction below the prior $E[f(X)]$ and favoring the default action. This indicates that the policy avoids aggressive exploration when nearby novelty is scarce. At PC 213 with $Seq_{Tx} = 1$, the same features become large and positive, pulling the prediction toward the branch action. The policy therefore recognizes abundant nearby novelty in the second transaction and prioritizes that branch.

Corpus-level consistency. The SHAP summary in Fig. 12(d) shows a consistent trend: high values of $covnum_{\text{false}}$ and $covnumNew_{\text{false}}$ increase the likelihood of selecting the branch action, while low values suppress it. The transaction index and depth modulate this effect, since novelty in later transactions is often more valuable for multi-transaction exploits. Overall, both global and local views confirm that the policy primarily responds to branch-local novelty and transaction context, aligning with the intended behavior of the mixed reward.

H.3 Limitations and Future Improvements

Acceleration limits on secure contracts. RSymX learns to prioritize likely vulnerable paths, but if a contract is safe, proving the absence of bugs may still require exploring most relevant paths. Thus, the search work needed for secure contracts is not fundamentally reduced, while model inference introduces only millisecond-level overhead. For contracts with massive execution spaces, RSymX caps exploration by n_{paths} to curb state explosion. However, vulnerable paths may still be missed if they fall outside the budget. Users therefore need to choose n_{paths} according to the desired trade-off between detection effectiveness and efficiency.

```
contract Integeroverflow {
  function calsum(uint x) public returns (uint){
    uint c = 0;
    for(uint i=1; i<=x; i++){
      c = c + i;   ← Overflow after >2^128 loops
    }
    return c;
  }
  function sum(uint[] x) public returns (uint){
    uint c = 0;
    for(uint i=0; i<x.length; i++){
      c = c + x[i];   ← Overflow after 2 loops
    }
    return c;
  }
}
```

Listing 7: The special examples of IO vulnerability.

False negatives on long paths. Because RSymX faithfully executes opcodes to preserve concrete state changes, excessively long paths or symbolically bounded loops may exceed time limits and cause false negatives. For example, dynamic arrays such as `sum(uint[] x)` in Listing 7 may exhaust gas or analysis time when the array length is unbounded. Bounding the loop length can reduce the check to a small number of arithmetic cases and expose some integer-overflow bugs, but vulnerabilities that become reachable only after extremely many iterations may still be missed. For instance, `calsum` contains an IO bug that requires more than 2^{128} loop iterations, making it impractical for path-by-path execution. Future work will explore loop summarization and solver-side closed-form arithmetic reasoning to reveal such conditions without explicit iteration.

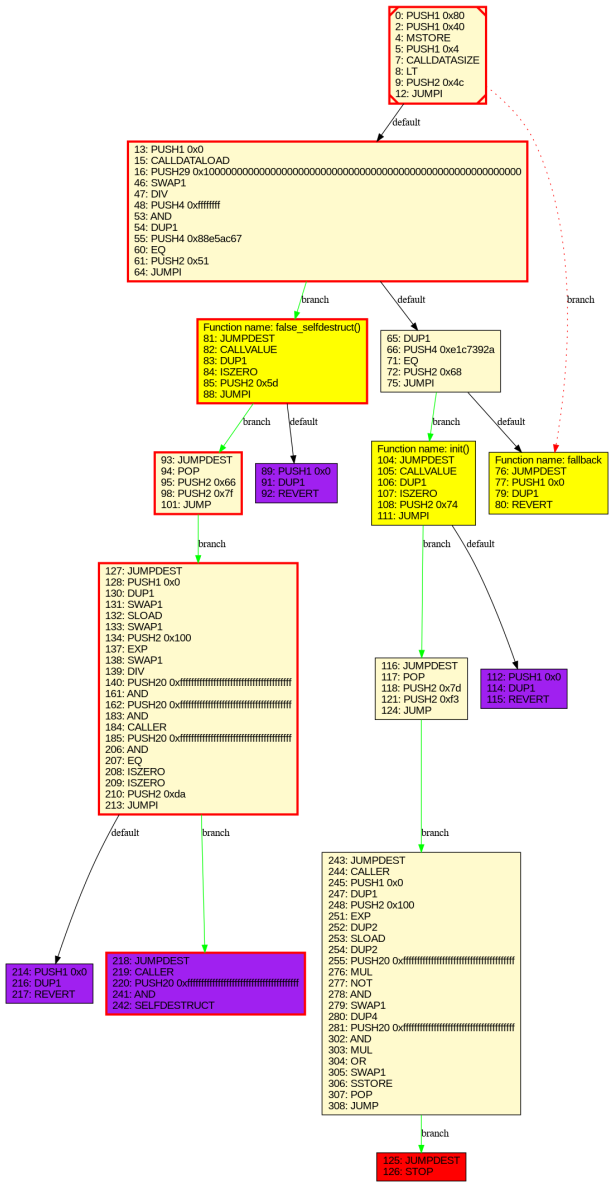


Figure 24: The CFG visualization of the *Suicidal* contract.

Solver optimization. SMT solvers such as Z3 [14] are central to symbolic analysis, but remain a bottleneck on complex path constraints, especially non-linear arithmetic. Practical improvements include constraint preprocessing, early UNSAT pruning, incremental solving, and adaptive timeout control [42, 62, 83]. Currently, RSymX uses fixed solver timeout settings; future versions can adjust these settings dynamically according to path complexity and solver feedback.

ABI-aware call-buffer construction. For bytecode-only contracts, RSymX builds a symbolic call buffer with a fixed length to invoke contract functions. Although this enables exploit-sequence generation without source code, overlong buffers introduce redundant symbolic bytes and slow solving, while underlong buffers may miss real function parameters. When ABI or API metadata is available, the call-data length can be specialized to each function, improving both accuracy and solving efficiency.

The Application Prospect of RSymX. For the venue of symbolic engineering, RSymX offers a risk-aware, verifiable analysis pipeline: semantics-faithful execution coupled with a mixed reward (time-normalized coverage and severity-weighted discovery) and a lightweight agent prioritizes high-impact traces under fixed budgets. Its compact design and modular detectors enable lightweight

IDE/CI integration and admission-control-style deployment, making it practical to adopt and enabling standardized cross-tool evaluation. Furthermore, with the rise of security-as-a-service, RSymX can integrate into developer workflows, such as providing Remix-based audits (as Slither does), or ship as a VSCode extension/Blockchain Development Kit component. These deployments bring continuous feedback to developers while strengthening runtime security guarantees.