Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing

Jianzhong Su
Sun Yat-sen University & Ant Group
sujzh3@mail2.sysu.edu.cn

Hong-Ning Dai
Hong Kong Baptist University
hndai@ieee.org

Lingjun Zhao
Sun Yat-sen University
zhaolj23@mail.sysu.edu.cn

Zibin Zheng*
Sun Yat-sen University
zhzibin@mail.sysu.edu.cn

Xiapu Luo
The Hong Kong Polytechnic University
csxluo@comp.polyu.edu.hk

ABSTRACT

As computer programs run on top of blockchain, smart contracts have proliferated a myriad of decentralized applications while bringing security vulnerabilities, which may cause huge financial losses. Thus, it is crucial and urgent to detect the vulnerabilities of smart contracts. However, existing fuzzers for smart contracts are still inefficient to detect sophisticated vulnerabilities that require specific vulnerable transaction sequences to trigger. To address this challenge, we propose a novel vulnerability-guided fuzzer based on reinforcement learning, namely RLF, for generating vulnerable transaction sequences to detect such sophisticated vulnerabilities in smart contracts. In particular, we firstly model the process of fuzzing smart contracts as a Markov decision process to construct our reinforcement learning framework. We then creatively design an appropriate reward with consideration of both vulnerability and code coverage so that it can effectively guide our fuzzer to generate specific transaction sequences to reveal vulnerabilities, especially for the vulnerabilities related to multiple functions. We conduct extensive experiments to evaluate RLF's performance. The experimental results demonstrate that our RLF outperforms state-of-the-art vulnerability-detection tools (e.g., detecting 8%-69% more vulnerabilities within 30 minutes).

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Fuzzing, Smart contract, Reinforcement learning

ACM Reference Format:
Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22) October 10-14, 2022, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3560429

1 INTRODUCTION

Blockchain technologies have attracted extensive attention from both industry and academia. As Turing-complete programs run on blockchains, smart contracts can achieve autonomous and trusted execution of user-defined business logic in the decentralized computing environment. Many popular applications have been implemented in smart contracts and running on Ethereum, the biggest blockchain platform that supports smart contracts, such as non-fungible tokens (NFT) [3], Decentralized finance (DeFi) [33], etc.

However, the proliferation of smart contracts has also exposed a myriad of security issues, especially the vulnerabilities [38]. The vulnerabilities may result in huge financial losses since most applications of smart contracts are involved digital assets. Therefore, it is crucial to reveal vulnerabilities for ensuring the security of smart contracts. Despite the advent of recent vulnerability-detection tools in smart contracts [16, 29], they become less effective for identifying complicated vulnerabilities with the increased complexity of smart contracts. One of the fundamental reasons lies in the stateful nature of smart contracts in contrast to traditional computer programs. In particular, the execution of smart contracts is involved with transaction sequences as input while maintaining *persistent states.* Thus, it may require specific transaction sequences to trigger non-trivial vulnerabilities [31] (a more concrete example to be given in § 3). Moreover, given the fact that there are countless possible transaction sequences for a smart contract, it becomes even more challenging to identify critical sequences (i.e., vulnerable transaction sequences) to trigger vulnerabilities.

In recent years, pioneer researchers have developed several vulnerability-detection tools to reveal vulnerabilities in smart contracts [16, 29]. Among these approaches, *fuzzing* has commonly been used to generate transactions as input to test smart contracts for vulnerability detection owing to its efficiency and conceptual simplicity. Although a number of fuzzing tools (a.k.a. fuzzers) for smart contracts have been developed, most of them cannot generate proper transaction sequences for triggering sophisticated vulnerabilities. Specifically, most fuzzers for smart contracts focus on improving the code coverage during the testing stage because of

[*] The corresponding authors.

ASE 2022, Oct. 10-14, 2022, Ann Arbor, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560429

the assumption that a higher code coverage means more vulnerabilities to be found. We call them as *coverage-guided* fuzzers. However, simply increasing code coverage does not necessarily increase the number of vulnerabilities being found [7]. Particularly, coverage-guided fuzzers may waste a lot of time generating non-vulnerable transaction sequences for achieving higher code coverage.

To address the aforementioned challenges, we present a novel *vulnerability-guided* fuzzer, namely the reinforcement learning fuzzer (RLF) for generating critical transaction sequences to detect vulnerabilities in smart contracts. Firstly, we model the process of fuzzing smart contracts as a Markov decision process (MDP) to construct our reinforcement learning framework since MDP can well handle the dynamic states of smart contracts during the fuzzing phase. Then, in order to guide our fuzzer to effectively generate vulnerable transaction sequences, we design an appropriate reward in RLF with consideration of both vulnerability and code coverage. Specifically, the reward of vulnerability is the key to guiding the fuzzer to generate vulnerable transaction sequences. However, purely using the reward of vulnerability may lead fuzzers to fall into local optimum and generate the transaction sequences that only contain the single vulnerable function. In this situation, the fuzzer would fail to reveal the vulnerabilities that need to call multiple functions to trigger. To this end, we integrate code coverage into the reward of vulnerability to steer our fuzzer to explore the vulnerable transaction sequences that involve multiple functions. In addition, we construct state space and action space, both of which are simple yet efficient for our RLF. Finally, the proposed RLF can effectively generate vulnerable transaction sequences for unseen contracts through learning from previous test (transaction) sequences.

We conduct extensive experiments to evaluate RLF. Comparing RLF with four advanced tools, Mythril [25], Smartian [11], SmarTest [31] and ILF [15], we find that our RLF outperforms existing tools in terms of the number of detected vulnerabilities within a limited time.

To summarize, our main contributions are as follows:

- We propose a new deep reinforcement learning framework for fuzzing smart contracts. It can handle the transformed states of runtime contracts and effectively generate vulnerable transaction sequences.
- We design a new reward with consideration of both vulnerability and code coverage to enhance vulnerability detection, especially for the complicated vulnerabilities related to multiple functions.
- We implement RLF and conduct extensive experiments on real-world smart contracts to evaluate its performance. The experiment results validate the effectiveness of our designed reward and other mechanisms. Moreover, the results also demonstrate that the proposed RLF can detect more vulnerabilities than other existing tools within a limited time.

The remainder of this paper is organized as follows. § 2 gives the background related to our approach. § 3 presents challenges of generating vulnerable transaction sequences, which motivate our study. § 4 elaborates on the detailed design of RLF. § 5 presents an extensive experimental evaluation of RLF. § 6 discusses the limitations of our RLF and outlines potential future improvement. § 7 surveys the related work. § 8 finally concludes this work.

2 BACKGROUND

In this section, we briefly review smart contract, fuzzing, and deep reinforcement learning.

2.1 Smart Contract

In this paper, we mainly focus on smart contracts in Ethereum. A smart contract is a Turing-complete program running on top of blockchain, which is the storage of smart contracts. Most Ethereum smart contracts are written by Solidity [5], which is a programming language specifically designed for smart contracts. The entire life cycle of a smart contract is described as follows. Before being deployed to Ethereum, source smart contracts written in Solidity are firstly compiled into bytecode, which has a one-to-one mapping to its opcode[1]. Once deployed, the bytecode is stored on blockchain and cannot be modified anymore. The smart contract can be invoked by sending a transaction. In particular, senders can specify which function to call and what parameters to input. Once receiving the transaction, the smart contract will execute the corresponding bytecode. After the transaction execution, the execution results are recorded on blockchain; otherwise, the execution results would be reverted. Although smart contracts cannot be modified once deployed, users can call the self-destroy function (e.g., `selfdestruct()`) to destroy the smart contract.

2.2 Fuzzing

Among a myriad of off-the-shelf software-testing tools, *fuzzing* is one of the most popular techniques because of its extensive empirical evidence in discovering real-world software vulnerabilities [21]. The main idea of fuzzing is to generate various inputs to run a computer program repeatedly and catch the information during the execution to analyze its vulnerabilities. In the emerging field of smart contracts, fuzzing has also received extensive attention [15, 17, 26]. Similar to computer programs, fuzzers generate transactions (i.e., inputs) to execute smart contracts and capture the execution logs, including the executed opcodes, the execution results, and other execution information during the testing phase. Based on the execution logs, in-depth analysis, such as detecting vulnerabilities [32] and evaluating performance [10], could be performed on smart contracts.

2.3 Deep Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm to empower agents to learn from the environment and take actions accordingly so as to achieve the maximized cumulative reward. As a foundation of RL, MDP is crucial to model the dynamics of the environment. In particular, MDP is essentially a discrete-time stochastic control process. At each time step, regarding the environment in any state denoted by s, the agent can choose any available action a. At the next time step, the environment moves into a new state s' and gives a corresponding reward r to the agent. To maximize the cumulative reward of the agent, instead of training on labeled data, RL learns from the previous experience obtained by some exploring strategies, such as the ϵ-greedy method. In practice, it is difficult for traditional RL methods to handle the explosive state space of smart

[1] An opcode contains some machine instructions to specify operations.

```
1  contract Crowdsale {
2    uint256 raised, closeTime, goal, status;
3    address owner;
4    mapping(address => uint256) deposits;
5
6    constructor(uint256 goalFund) public {
7      closeTime = now + 30 days;
8      owner = msg.sender;
9      goal = goalFund;
10     status = 0;
11     raised = 0;
12   }
13
14   function setStatus(uint256 newStatus) public {
15     require((newStatus == 1 && raised >= goal) ||
16             (newStatus == 2 && raised < goal
17               && now >= closeTime));
18     status = newStatus;
19   }
20
21   function setOwner(address newOwner) public {
22     // Debug: require(msg.sender == owner);
23     owner = newOwner;
24   }
25
26   function invest() public payable {
27     require(status == 0 && raised < goal);
28     deposits[msg.sender] += msg.value;
29     raised += msg.value;
30   }
31
32   function withdraw() public {
33     require(status == 1);
34     owner.transfer(raised);
35   }
36
37   function refund() public {
38     require(status == 2);
39     // bug(); some operations with bug
40     msg.sender.transfer(deposits[msg.sender]);
41     deposits[msg.sender] = 0;
42   }
43 }
```

Figure 1: A Crowdsale contract (simplified version) containing vulnerability that allows attacker to steal funds.

contracts. To address this challenge, we employ deep reinforcement learning (DRL), i.e., the integration of deep neural network (DNNs) with traditional RL. DRL is quite promising to address the explosive state space of fuzzing in smart contracts.

3 MOTIVATION

We first use a motivating example to illustrate why generating a proper transaction sequence is important for uncovering sophisticated vulnerabilities, and then discuss the challenges in the generation of such transaction sequences.

3.1 Motivating Example

We consider a Crowdsale contract as our motivating example. It was used to gather funds with a goal to collect a certain amount of ethers (i.e., Ethereum tokens) within a period (e.g., 30 days). The crowdsale is successful if this goal is reached within the given period. Consequently, the crowdsale owner can withdraw the funds. Otherwise, the crowdsale fails and investors can refund the ethers. Fig. 1 depicts a simplified version of the Solidity code of the Crowdsale contract. Firstly, the owner deploys the contract while initializing `goal` (at line 9). Then, users call `invest()` to send ethers to this contract. Once the crowdsale succeeds, the owner of the contract can call `setStatus()` to change `status = 1` (at line 18) and get the funds from the contract by calling `withdraw()`.

This contract contains an Ether-Leaking vulnerability, which allows an attacker to steal the funds from the Crowdsale contract. In particular, when the following vulnerable transaction sequence is conducted, this vulnerability will be triggered.

(1) Users call `invest()` and send enough ethers to make the raised fund ≥ `goal` (at line 29);
(2) The attacker calls `setOwner()` to set `owner` to the attacker's address (at line 23);
(3) The attacker calls `setStatus()` to set `status` to 1 (at line 18);
(4) The attacker calls `withdraw()` to steal the funds from the contract (at line 34).

As a result, the attacker becomes the owner of the contract and takes away the funds. Since a specific transaction sequence is necessary to trigger this vulnerability, most existing smart contract fuzzers may fail to generate the corresponding transaction sequences to reveal this sophisticated vulnerability. In addition, if there exists `bug()` in `refund()` (line 39), the investors cannot retrieve their funds when the crowdsale fails. This potential vulnerability also needs specific vulnerable transaction sequences to trigger.

3.2 Challenges to Generate Vulnerable Transaction Sequences

Intrinsic Features of Smart Contracts. More precisely, the most important feature of smart contracts is *statefulness* [11, 15]. In particular, a smart contract saves its state in blockchain. A transaction sequence may frequently change the contract's state, e.g., the variable `raised` is frequently changed by function `invest()` in the previous mentioned Crowdsale contract. Thus, smart contracts are sensitive to the state alterations caused by transactions. As a result, different transaction sequences will cause diverse states in smart contracts and multiple triggering conditions. For example, the owner cannot withdraw the funds from the contract unless the variable `status` is set to 1. In this situation, a specific transaction sequence is necessary to set `status` to 1. Therefore, to trigger the Ether-Leaking vulnerability in the Crowdsale contract, a specific vulnerable transaction sequence is needed. However, among the countless possible transaction sequences in smart contracts, it is difficult to find out the vulnerable transaction sequences.

Unfortunately, it is challenging for existing fuzzers to tackle this challenge because of their limitations. For example, some fuzzers, such as ContractFuzzer [17] and sFuzz [26], mainly focus on generating specific inputs to satisfy complex conditions (e.g., the statement `require`). However, these fuzzers ignore the order of transaction sequences, which is the key to revealing the complicated vulnerabilities as shown in the Crowdsale contract. It is hard for these fuzzers to generate vulnerable transaction sequences.

Limitation of Coverage-guided Approaches. Some fuzzers support handling the transaction sequences for more effective testing.

Imitation Learning-based Fuzzer (ILF) [15] utilizes imitation learning to learn from symbolic execution for exploring deep execution paths. SMARTIAN [11] employs both static and dynamic analyses, and mutates the input based on data-flow coverage. Both these two fuzzers are coverage-guided fuzzers with the goal of maximizing the code coverage of smart contracts during fuzzing. Nonetheless, in the case of vulnerability detection, higher code coverage does not necessarily lead to finding more vulnerabilities [7]. As indicated in [12], only a few functions in smart contracts contain vulnerabilities while most of them contain no vulnerability. When there are multiple execution paths in a smart contract, different transaction sequences may explore different execution paths but only a few sequences can trigger vulnerabilities. In other words, coverage-guided fuzzers may not be efficient in generating transaction sequences to trigger vulnerabilities since they only pursue higher code coverage and waste too much time exploring the sequences without triggering vulnerabilities. For example, if a contract is composed of functions `fun_A` (vulnerable, containing 10 lines) and `fun_B` (not vulnerable, containing 90 lines), coverage-guided fuzzers may consume lots of time on `fun_B` for higher code coverage but fail to trigger the vulnerability in `fun_A`.

Limitation of Vulnerability-guided Approaches. To increase the efficiency of vulnerability detection, vulnerability-guided approaches are proposed. SMARTEST [31] utilizes language models to guide symbolic executor to generate vulnerable transaction sequences. Although SMARTEST attempts to reveal as many vulnerabilities as possible, it may be stuck in the local optimum and only hunt the vulnerable transaction sequences related to a single function (i.e., the vulnerability can be triggered by only testing one function). For example, we conduct a set of experiments on SMARTEST and find that it fails to reveal the aforementioned Ether-Leaking vulnerability within 30 minutes. The failure lies in the vulnerable transaction sequences related to four functions but SMARTEST cannot explore the vulnerable transaction sequences with too many functions. Thus, only considering vulnerability information (by *pure* vulnerability-guided approaches like SMARTEST) is not beneficial to generating the vulnerable transaction sequences related to invocation across multiple functions.

Both the intrinsic features of smart contracts and the limitations of existing vulnerability-detection tools motivate us to design an advanced tool to generate vulnerable transaction sequences to address the above challenges.

4 THE RLF SYSTEM

In order to address the above-mentioned challenges, we propose an advanced vulnerability-guided fuzzer to generate vulnerable transaction sequences based on reinforcement learning. We first model the process of fuzzing smart contracts as an MDP so that we can construct a reinforcement learning framework for generating transaction sequences. Then, we give an overview of our approach, namely RLF, and describe the detailed design of each of its core components.

4.1 Modeling Fuzzing as MDP

MDP can model a sequence of decisions (reactions) done by an agent to an environment. It consists of an *agent* who makes a decision to the *environment* and correspondingly earns a *reward*. Accordingly, we model the process of fuzzing smart contracts as an MDP.

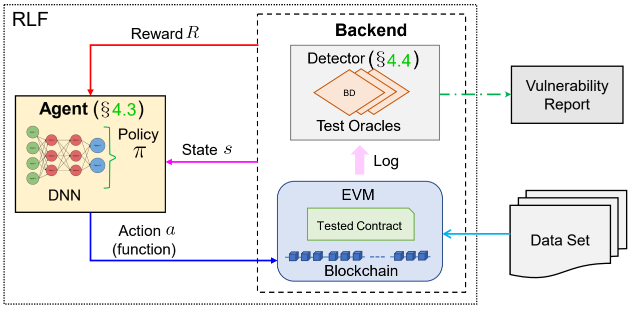

Figure 2: Overview of RLF. At each step, Agent selects an action to Backend, Backend returns the state and reward to Agent.

In the process of fuzzing smart contracts, we generate transaction sequences for fuzzing the tested contract. A transaction consists of the function to be invoked, the parameters of the function, and other elements. As shown in the example in § 3.1, the function-call sequence within the transaction sequence plays a decisive role in the effect of detecting vulnerabilities. Consequently, we convert the generation of transaction sequences to that of function-call sequences; during this process, other elements of the transaction are filled randomly from our seed pools.

We define $s(SC, \bar{f})$ as the state of Agent, where s contains the features of the tested contract SC and previous function-call sequence $\bar{f} = \{f_1, f_2, ..., f_n\}$, which contains each function executed by the tested contract. At each step, Agent selects f by its policy π as the new action a, which is expressed as follows,

$$a \leftarrow \pi\big(s(SC, \bar{f})\big), \quad a \in A, \quad s \in S,$$

where A and S denote the action space and the state space, respectively. After that, the function f is packaged as a transaction and the tested contract executes the transaction in the backend to change the state. Thus, the old state s is updated to the new state s' while Agent receives reward r. This process is depicted as follows:

$$(s', r) \leftarrow P\big(s(SC, \bar{f}), a\big), \quad a \in A, \quad s, s' \in S,$$

where $P\big(s, a\big)$ is the transition function based on the tested contract and EVM. Our goal is to construct a policy π^* so as to obtain as much cumulative reward as possible within M steps:

$$\pi^* \leftarrow \arg\max_{\pi} \sum_{t=0}^{M} r_{t+1}^{\pi}.$$

Therefore, to generate vulnerable function-call sequences based on the above MDP, we need to design the following items.

(1) Suitable action a and state s for the policy (§ 4.3.1 and § 4.3.2);
(2) An appropriate reward r to effectively guide the policy to generate vulnerable function-call sequences (§ 4.3.3).
(3) A policy to maximize the reward, which can handle the transformed states of tested contracts during testing so as to generate specific function-call sequences (§ 4.3.4);

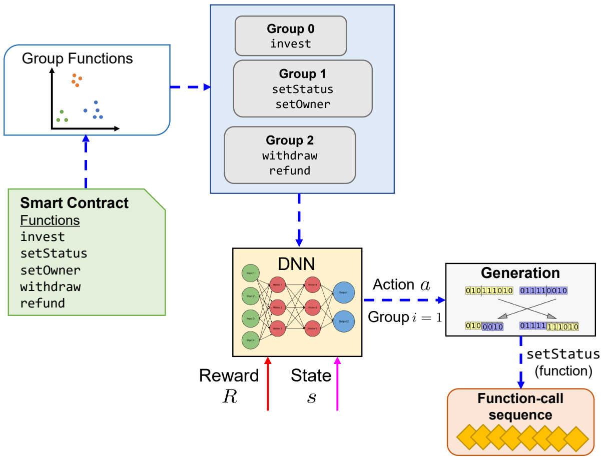

Figure 3: Working flow of Agent. At this step, DNN chooses Group 1 as Action and function setStatus is selected to execute tested contracts.

4.2 Overview of RLF

To determine the above items, we firstly design action *a* and state *s* according to the unique features of smart contracts. Secondly, we design an appropriate reward related to both vulnerabilities and code coverage for steering policy to generate vulnerable function-call sequences. Thirdly, we construct a deep neural network (DNN) as the policy to learn from the previous experience (i.e., transaction sequences) and generate specific function-call sequences to maximize the accumulated reward. Finally, we implement RLF.

Fig. 2 depicts the overview of RLF, which contains two major components: *Backend* and *Agent*. *Backend* includes EVM and Detector. EVM is used to execute smart contracts while Detector is used to detect vulnerabilities based on test oracles. When *Backend* receives an action, the tested contract executes the corresponding function (transaction) in EVM. The execution log is next exported to Detector, which exploits test oracles to detect vulnerabilities. According to the execution log and vulnerabilities being detected by Detector, *Backend* provides Agent with both state and reward. Agent utilizes the DNN to learn from previous sequences and select functions to execute tested contracts. After fuzzing, Detector reports the vulnerabilities being found in tested contracts. § 4.3 and § 4.4 present the details of Agent and Detector, respectively.

4.3 Agent

We further elaborate on the design of Agent, which generates function-call sequences for fuzzing. Fig. 3 shows the working flow of Agent. In particular, functions in the tested contract are categorized into groups (to be explained in § 4.3.1). At each step, DNN firstly chooses the best action (group) according to the state and then randomly selects a function from the group to execute tested contracts.

We first illustrate how to design the action and state of Agent, and then present the detail of the reward. Finally, we present the architecture of DNN, which is used to learn from previous test sequences and choose actions according to states at each step.

Table 1: Status operations in function.

Operation	Description
Payable	The function can receive ether from other addresses.
Call	The function can transfer ether or invoke other contracts.
Store	The function can change the storage.
Selfdestruct	The function can destroy the contract.

4.3.1 Action. As the output of Agent, the action has the action space, which is the feasible domain of actions. To generate function-call sequences, we need to select a function to call at each step. Thus, we define the action of Agent as the function selection and then model function space as discrete action space. Because of the variety of numerous functions in various smart contracts, we need to build a unified cross-contracts action space for training DNN (i.e., the action space of each contract is identical). Thus, we need to classify the functions into several groups to simplify the function space. Specifically, rather than directly selecting functions, Agent selects a function group as the action and then randomly chooses a function from the group. Next, we illustrate the details of function classification as follows.

Function classification. Note that the function-call sequences to trigger vulnerabilities are always related to the persistent status of smart contracts (e.g., balance and its relevant variables). For example, the precondition for the Ether-Leaking vulnerability in Fig. 1 to be triggered is that both variable `raised` (i.e., the balance of Crowdsale contract) and `status` meet certain conditions. Several related functions need to be called for changing the status of the Crowdsale contract to fulfill certain conditions. Accordingly, it is important to handle the persistent status of smart contracts while constructing the function-call sequences to trigger vulnerabilities. We classify functions according to the types of status operations. Table 1 summarizes the types of status operations.

In general, smart contracts have three types of statuses: *balance*, *storage* and *existence*. *Balance* means the number of ethers owned by the smart contract. *Storage* means the variables stored in blockchain, which persist across transaction sequences. *Existence* means that the smart contract has not been destructed. To change the status of a smart contract, we need to invoke the corresponding status operation. For the *balance* operation, both *Payable* and *Call* can change the balance of smart contracts. *Payable* reflects the function that can receive the ether from other addresses. *Call* indicates that the function can transfer ether to other addresses, which is the potential basis of some vulnerabilities related to the ether transfer, such as the Ether-Leaking vulnerability. In addition, *Call* also means that the function can invoke other smart contracts, thereby producing more complicated behaviors. For the *storage* operation, *Store* implies that the function can operate the storage. For the *existence* operation, *Selfdestruct* means that the function can destroy the contract. Through this way, users can specify an address to receive the ether when destroying the contract.

We take the functions in the Crowdsale contract as examples to elaborate the above status operations. In particular, `invest()` is the only *Payable* function while both `withdraw()` and `refund()` are *Call* functions. Regarding *storage* operations, `setOwner()` is *Store*

Table 2: State of Agent. f is the last call function in $\bar{f}$.

	Features	Description
SC	*Action*	Frequency of each action.
	Trace	Proportion of 8 key opcodes being executed cumulatively.
	Coverage	Instruction and block coverage of tested contract.
F	*Revert*	Fraction of ending with `revert` when executing f.
	Return	Fraction of ending with `return` when executing f.
	Assert	Fraction of ending with `assert` when executing f.
	Call	Frequency of f being executed in $\bar{f}$.
	Coverage	Instruction and block coverage of function f.
	Arguments	Number of arguments of f.
	Opcodes	Counts of 50 representative opcodes in function f.
	Name	Word embedding of f's name.

function. There are no *Selfdestruct* functions in the Crowdsale contract. To construct the function-call sequences to trigger the Ether-Leaking vulnerability in the Crowdsale contract, we should firstly select the function group with *Payable* status operation. We then select the function group with *Store* status operation to change the storage of the contract. Finally, we select the function group with *Call* status operation to withdraw the funds in the contract.

In particular, if a function does not have these four status operations, the function would be a *pure*/*view* function [5] that has no side effects on the status of the smart contract. Since pure/view functions cannot handle the status of smart contracts, it is impossible to trigger vulnerabilities by directly calling these functions. we can improve the efficiency of our fuzzer by dropping these pure/view functions from our action space.

Based on these status operations, we can classify the functions into multiple groups. Since the functions in a group have the same type of status operations, Agent can select the function with the specific type of status operations via the selection of the corresponding function group at each step. Grouping functions by status operations is a straightforward way to simplify the function space and works efficiently for our RLF. At each step, Agent firstly selects a group as the action, and then randomly chooses a function from it. Meanwhile, we build up seed pools in a similar way to ILF [15] and randomly generate inputs (arguments) according to Application Binary Interface (ABI) including input structural information.

4.3.2 State. The state is the input of Agent, the state space represents all possible states of Agent. Due to the numerous states of smart contracts, we construct a continuous state space (denoted by a continuous vector) to comprehensively represent the state of Agent. We extract features from the contract *SC* and previous function-call sequence $\bar{f}$ as the state, as given in Table 2.

Feature *SC*. The features extracted from *SC* capture the execution history representing the dynamic status of the entire tested contract during testing. Feature *Action* counts the frequency of each action to record the action history of Agent. Feature *Trace* records the proportional eight key opcodes (i.e., sha3, call, create, selfdestruct, jump, jumpi, jumpi, sload and sstore), which are cumulatively executed by the tested contract. These eight opcodes are related to logical jump, storage operation, and ether transfer. Meanwhile, feature *Coverage* represents the code coverage of the tested contract during testing.

Table 3: Description of different rewards.

Reward	Description
R_{block}	The fraction of the executed basic blocks to all basic blocks.
R_{cov}	$\frac{1}{N}\sum_{i=1}^{N}\left(\frac{1}{M_i}\sum_{j=1}^{M_i} R_{\text{block}}(f_j)\right)$
R_{bugs}	1 if **Detector** identifies any vulnerabilities; otherwise 0.
R_{mix}	$\alpha R_{\text{bugs}} + (1-\alpha)R_{\text{cov}}$.

Feature *F*. The features extracted from previous function-call sequences capture the semantic features of the last call function f in $\bar{f}$. The top-three features record important information about the success or the failure of the last call to f. For example, feature *Revert* measures the fraction of ending with revert when executing f. A large value of *Revert* means that many calls to f in $\bar{f}$ revert. This is because the complex precondition of f is difficult to satisfy, and f requires more calls to fully test. Feature *Call* measures the frequency of f being executed. Feature *Coverage* captures the coverage of f.

The last three features capture the static properties of f. Feature *Arguments* measures the number of arguments to be input to the function f. Specifically, more arguments mean the more complex function f. Feature *Opcodes* counts 50 representative opcodes in function f for measuring the functionality and complexity of f. We select these 50 opcodes with reference to [15]. In addition, feature *Name* encodes the function's name to capture the semantic meaning of functions. Specifically, we first tokenize f's name into separate sub-tokens and map each sub-token into *word2vec* word embedding [22], and then average the embeddings to obtain the final embedding, a 300-dimensional vector, as feature *Name*. Particularly, to reduce the dimension of features, we utilize a pre-trained network [15] to compress the above features as F.

At each step, we merge the aforementioned features (SC, F) as the state of Agent.

4.3.3 Reward. The reward essentially determines the goal of Agent. We design a new appropriate reward that can guide Agent to generate vulnerable function-call sequences. In order to trigger the potential vulnerability, it is natural to design a reward that is related to the vulnerabilities. In other words, a higher cumulative reward means more vulnerabilities are found. In this way, the goal of Agent with the maximized cumulative reward would guide Agent to generate vulnerable function-call sequences. We denote the reward of vulnerabilities by R_{bug}, which is a positive value when the test oracles identify vulnerabilities; is zero otherwise. Based on the reward R_{bug}, we build up a vulnerability-guided function-call sequences generator to reveal smart-contract vulnerabilities similar to SmarTest [31].

However, as we discuss in § 3.2, purely using the reward of vulnerabilities may lead Agent to fall into local optimum and focus on the vulnerabilities related to a single function. In this case, all function calls in the sequence only concentrate on a single vulnerable function. As a result, the fuzzer is struggling to generate vulnerable function-call related to multiple functions. To tackle this problem, we integrate the code coverage reward into the reward of vulnerabilities so that Agent has the ability to explore the vulnerable function-call sequences containing multiple functions. Therefore, we need a reward of code coverage for each function group (action) to let Agent avoid only choosing the function group that contains the vulnerable function. In particular, we use the *block coverage* as the reward of code coverage denoted by R_{block}, which is the fraction of the executed basic blocks to all basic blocks[2]. We firstly count the block coverage of each function group (action) according to the classification in § 4.3.1, and then take their average as our reward of code coverage, which is expressed as follows:

$$R_{\text{cov}} = \frac{1}{N}\sum_{i=1}^{N}\Big(\frac{1}{M_i}\sum_{j=1}^{M_i} R_{\text{block}}(f_j)\Big), \tag{1}$$

where N is the number of function groups (actions) and M_i is the number of functions in the i-th function group. In this way, we can use R_{cov} to represent the block coverage of contracts while reducing the influence of the unbalanced number of functions on different groups. Finally, we propose an appropriate reward that effectively guides Agent to generate specific function-call sequences for revealing the vulnerabilities related to multiple functions. The reward is expressed as follows:

$$R_{\text{mix}} = \alpha R_{\text{bugs}} + (1-\alpha)R_{\text{cov}}, \tag{2}$$

where α is an adjustment factor. The proposed reward considers both the vulnerability and code coverage (given in Eq. (1)). Table 3 summarizes all the above rewards.

Algorithm 1: Workflow of Training RLF.

```
input  : Iteration steps $T$, Sample size $M$, Search rate $\epsilon$,
         Episode $E$, Smart Contract $SC$
output : Network $Q$
$Q \leftarrow$ InitializeNetwork(); $s \leftarrow$ getInitState();
$A, G \leftarrow$ groupFunctions($SC$);
for $k \leftarrow 1$ to $T$ do
    $a \leftarrow$ getRandomOrBestAction($A, Q, \epsilon$);
    $f(x) \leftarrow$ selectFunction($G[a]$);
    $s', r \leftarrow$ executeFunction($f(x), SC$);
    saveExperience($s, a, r, s'$);
    $s \leftarrow s'$;
    if $k$ MOD $E == 0$ then
        $\{(s_i, a_i, r_i, s'_i)\} \leftarrow$ loadExperience($M$);
        $Q \leftarrow$ train($\{(s_i, a_i, r_i, s'_i)\}, Q$);
    end
end
```

4.3.4 Neural Network of Agent. According to the continuous state space and the discrete action space as described above, we use *Deep Q-learning Network* (DQN) [23] to construct the reinforcement learning network for fuzzing since DQN has strengths in efficiently handling a large state space. However, DQN is limited in the sense that they learn from a limited number of past steps, and the function-call sequence is too long for DQN to learn. To address this issue, we add *recurrency* to DQN by replacing a linear layer with a recurrent LSTM, and the new network called *Deep Recurrent Q-Network* (DRQN) [14].

Table 4: Test oracles for vulnerabilities

ID	Bug	Description
EL	Ether Leaking	The contract can be stolen ethers by attackers.
SC	Suicidal Contract	The contract can be destroyed by attackers.
BD	Block State Dependency	The contract transfer ethers depends on block state variable (e.g., timestamp, blocknum).
UE	Unhandled Exception	The contract does not check the return of external call.
DD	Dangerous Delegatecall	The contract allow attacker to arbitrarily inject the arguments of delegatecall.
EF	Ether Frezzing	The contract can only receive ethers but does not contain create, call, delegatecall and selfdestruct opcodes (only ways to send out ethers).

Algorithm 1 describes the complete workflow of training RLF. At each step, we use the ϵ-greedy strategy to choose the action (i.e., selecting the action randomly with ϵ probability or selecting the action by Agent) and randomly call the function according to the action (i.e., the function group). After the function is executed, we save the experience (s, a, r, s') so as to replay the experience to update the network Q. In particular, the randomness of RLF may impact the learning component. For example, a function-call sequence may trigger vulnerabilities but the fuzzer assigns it with bad parameters. However, these kinds of sequences do not influence updating neural network since they have no reward. In our RLF, only the function-call sequences with proper parameters can trigger vulnerabilities and then can obtain the reward, consequently updating the neural network. As a result, the neural network can generate these kinds of function-call sequences in the testing phase.

4.4 Detector

To detect vulnerabilities, inspired by [9, 15], we set up six test oracles, which are listed in Table 4 with corresponding brief descriptions. Specifically, we extract the execution log (e.g., executed opcodes) of the tested smart contract during fuzzing and apply the test oracles to identify whether the vulnerabilities are triggered.

5 EVALUATION

In this section, we conduct extensive experiments to evaluate our RLF by addressing the following questions.

- **RQ1**. How does the reward impact RLF's performance? (§ 5.3)
- **RQ2**. Does RLF detect vulnerabilities more efficiently than existing state-of-the-art tools? (§ 5.4)
- **RQ3**. How does RLF perform on real-world smart contracts? (§ 5.5)

5.1 Experiment Setup

Implementation of RLF. We run RLF on a modified backend based on [15] that supports fast executing transactions natively without performing Remote Procedure Call (RPC). On the top of Pytorch [4], we implement the neural network that consists of three linear layers and one LSTM layer. In the training phase, we assign ϵ with a large value so that Agent can explore as many states and actions as possible while in the testing phase, we decrease ϵ to 0.15 so that Agent can utilize the trained neural network to generate

[2] A basic block is a sequence of straight-line opcodes without in-branches except for the entry and without out-branches except for the exit.

Table 5: Datasets Used.

ID	Source	Used For	Avg. LoC	Num. of Contracts
D1	VeriSmart [2]	RQ1, RQ2, RQ3	330	85
D2	XBlock [6]	RQ3	343	1206

Table 6: The number of functions in each group (action) that are classified by status operations. The function group with • (○) means the functions contain (not contain) the status operations. The function group with □ means the functions contain either status operations or no status operations.

	Payable	*Call*	*Store*	*Selfdestruct*	Number
Group 1	●	●	□	○	51
Group 2	○	●	□	○	242
Group 3	●	○	□	○	97
Group 4	○	○	●	○	659
Group 5	□	□	□	●	55
Pure/View Functions	○	○	○	○	929

vulnerable transaction sequences. In particular, the tested contract reverts to the initial state after every 50 steps to avoid getting stuck in a locked condition during fuzzing.

Comparing Tools and Configurations. We mainly consider the vulnerability-detection tools that are publicly available and can support transaction generation. These tools can be divided into two categories: symbolic execution and fuzzing. Regarding symbolic execution, there exist several well-known tools, such as Mythril [25], TEETHER [19] and SMARTEST [31]. For comparison, we select an industry tool Mythril, which has shown its powerful performance on vulnerability detection [12], and a vulnerability-guided symbolic executor SMARTEST [31]. With respect to fuzzing, the representative tools include ContractFuzzer [17], ILF [15], sFuzz [26], and SMARTIAN [11]. We choose ILF and SMARTIAN since they have superior performance to others. Moreover, all the experiments are conducted on a Ubuntu machine with an Intel(R) Core(TM) i9-10980XE (3.00GHz) CPU (36 cores and 72 threads in total), a GPU at 3090Ti, and 256 GB of memory.

Datasets. To conduct our experiments, we build two datasets. The first dataset is obtained from a labeled dataset [31], which contains Ether-Leaking and Suicidal Contract vulnerabilities. Note that they are typical vulnerabilities that need specific transaction sequences to trigger. Since each tool can detect vulnerabilities within different ranges, we only use 85 vulnerable contracts that can be successfully analyzed by every selected tool as our first dataset, namely **D1**. In these 85 vulnerable contracts, there are 108 functions with Ether-Leaking and 46 functions with Suicidal Contract vulnerabilities. Particularly, some contracts in dataset **D1** are created or modified manually. Thus, to evaluate the effectiveness of each tool on real-world contracts, we build up the second dataset from the smart contracts deployed on the Ethereum mainnet. To satisfy the running requirements of all the tools, we choose the smart contracts with 0.4.25 version and collect them from XBlock [37] and remove the duplicates. After this process, we randomly sample 1,206 smart contracts as the second dataset, namely **D2**. Table 5 summarizes these two datasets.

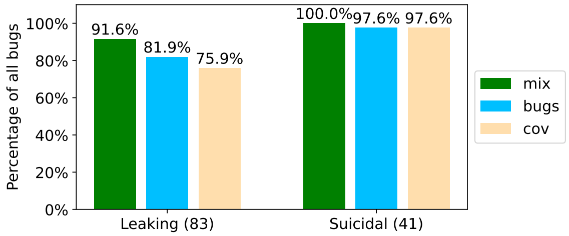

Figure 4: Detected vulnerabilities by fuzzers with different rewards. The number of all vulnerabilities is computed as the union of all vulnerabilities detected by each fuzzer.

5.2 Distribution of Functions in Actions

Before answering the RQs, we illustrate how to group the functions as the action space for RLF, and briefly discuss the distribution of the functions in each action.

We collect all the functions from dataset **D1** and finally obtain 2,033 functions. Based on the four status operations specified in § 4.3.1, we try several combinations to group the functions and conduct small-scale experiments for evaluation. Finally, we classify the functions into six groups and present the results in Table 6. These function groups have different combination schemes of status operations. For example, the functions in Group 1 have *Payable* and *Call* operations while having no *Selfdestruct* operation. Meanwhile, they have or have no *Store* operation. Particularly, there is no intersection of different function groups.

It is worth noting that half of the functions are pure/view functions in smart contracts. Excluding these pure/view functions in our action space can improve the efficiency of fuzzing since these functions cannot mutate the status of smart contracts. Thus, we exclude the group with pure/view functions and use the rest five groups as our actions. We also consider this grouping scheme to construct actions for other smart contracts in dataset **D2**. Despite its simplicity, this grouping scheme is constructive to an effective action space for RLF without too many overheads.

5.3 RQ1: Impacts of Rewards

To answer RQ1, we conduct comparative experiments on RLF with different rewards in order to investigate the relationship between the rewards and the performance of vulnerability detection. In this group of experiments, we perform a standard 2-fold cross validation. Specifically, we randomly split dataset **D1** into two partitions. Each time we select one partition as the training data and another partition as the testing data. We repeat this training and testing procedure two times on the different splits and obtain testing results on the entire dataset **D1**. For each reward, RLF runs for 2,000 steps on each contract in testing phase. Moreover, we let $\alpha = 0.7$ for R_{mix} (i.e., $R_{\text{mix}} = 0.7R_{\text{bugs}} + 0.3R_{\text{cov}}$). Fig. 4 presents the vulnerabilities being found by RLF with different rewards, R_{bugs}, R_{coverage} and R_{mix} in terms of the percentage of the number of detected vulnerabilities (bugs) to the total number of vulnerabilities, which is the union of all the vulnerabilities detected by each fuzzers.

We have the following observations from Fig. 4: 1) For Ether-Leaking (EL), RLF with R_{bugs} detects 6% more vulnerabilities than

Table 7: Vulnerabilities reported by different tools in D1. The number in ( ) is the number of functions with corresponding vulnerabilities. TP is the number of True Positives and # is the number of reported vulnerabilities.

	Tools	EL (108)		SC (46)		ALL (154)	
		TP	#	TP	#	TP	#
Symbolic Executors	**Mythril** [25]	9	12	26	26	35	38
	SmarTest [31]	62	63	41	41	103	104
Fuzzers	**Smartian** [11]	21	21	22	22	43	43
	ILF$_{\text{retrained}}$ [15]	78	78	41	41	119	119
	ILF [15]	86	88	43	43	129	131
	our RLF	98	100	43	43	141	143

that with R_{cov}, implying that the reward of vulnerabilities can effectively guide RLF to generate the vulnerable transaction sequences. 2) RLF with R_{mix} detects the most number of vulnerabilities, particularly, 9.7% more than that with R_{bugs} only. This is because the reward of the code coverage steers RLF to explore the vulnerable transaction sequences with multiple functions. As a result, the RLF with R_{mix} outperforms that with a single vulnerability reward R_{bugs}. 3) Compared to Ether-Leaking, Suicidal (SC) is relatively easier to detect, since it needs simpler transaction sequences to trigger. As a result, for Suicidal, RLF with R_{mix} only detects one more vulnerability than others. In summary, combining the vulnerability reward and the code coverage reward can effectively guide RLF to generate more vulnerable transaction sequences.

Answer to RQ1: R_{bugs} guides RLF to detect 6% more vulnerabilities than RLF with R_{cov}. When both R_{bugs} and R_{cov} are used together, R_{mix} enables RLF to detect the most number of vulnerabilities, i.e., 9.7% and 15.7% more than those with R_{bugs} and R_{cov}, respectively.

5.4 RQ2: Efficiency of RLF

In this experiment, we evaluate the efficiency of tools in detecting Ether-Leaking and Suicidal Contract vulnerabilities. We run Mythril, SmarTest, Smartian, ILF and RLF on **D1**. For Mythril and SmarTest, we let the analysis timeout be 30 minutes and the Z3 timeout be 90s per contract. For RLF, we let the reward of RLF be R_{mix} with $\alpha = 0.7$ ($R_{\text{mix}} = 0.7R_{\text{bugs}} + 0.3R_{\text{cov}}$) and perform a standard 2-fold cross validation as the experiments in § 5.3. Particularly, since ILF is also a learning-based fuzzer, we not only run ILF with the pre-trained neural network, but also retrain ILF with our dataset according to the settings in its GitHub source [1], named as ILF and ILF$_{\text{retrained}}$, respectively (for a fair comparison). In the testing phase, we run each fuzzer for 30 minutes per contract.

After automated detection, we identify the reported vulnerabilities according to the labels in **D1** and list the results in Table 7. Accordingly, we also plot the number of true vulnerabilities being found versus time (second) by different tools in Fig. 5. Within 30 minutes (1,800s), RLF demonstrates its efficiency in vulnerability detection and detects the most number of true vulnerabilities. We next compare RLF with other symbolic executors and fuzzers.

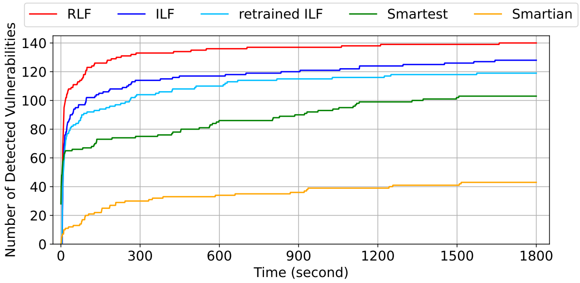

Figure 5: True vulnerabilities being found versus time (second) by different tools in D1.

Comparing RLF to Existing Symbolic Executors. Both RLF and SmarTest find much more true vulnerabilities than Mythril. This is because our RLF and SmarTest are vulnerability-guided approaches so that they focus on generating vulnerable transaction sequences rather than covering the non-vulnerable code for merely high code coverage. Comparing RLF to SmarTest, RLF detects 25% (i.e., $\frac{141-103}{154}$) more true vulnerabilities than SmarTest. This is because SmarTest fails to detect the vulnerabilities related to multiple functions (e.g., the Ether-Leaking vulnerability as shown in the example of Fig. 1). In addition, Fig. 5 shows that the number of vulnerabilities detected by SmarTest still increases rapidly in the last 900s (from 900s to 1,800s); it may imply that the number of its detected vulnerabilities has not achieved the maximum value even after a long time. This is mainly because the low efficiency of the symbolic execution mechanism (e.g., the SMT solver) limits the performance of SmarTest.

Comparing RLF to Existing Fuzzers. Comparing ILF$_{\text{retrained}}$ to ILF, ILF detects 6% (i.e., $\frac{129-119}{154}$) more vulnerabilities. According to the analysis of loss values during training, we find that the neural network of ILF cannot converge at the end of training. This is because the number of smart contracts in dataset **D1** is too small to train the complex neural network of ILF, which requires a huge smart-contract dataset to train. As a result, ILF$_{\text{retrained}}$ performs worse than ILF with the pre-trained network.

Next, we compare our RLF to ILF, RLF detects 8% (i.e., $\frac{141-129}{154}$) more true vulnerabilities. This is mainly because ILF is a coverage-guided fuzzer with an aim to cover as many execution paths as possible, inevitably leading to huge time consumption. By contrast, as a vulnerability-guided fuzzer, RLF mainly focuses on reaching vulnerable execution paths. Moreover, ILF consumes more time in calculating the inputs of functions while RLF simplifies this process by randomly generating the inputs. Specifically, ILF is composed of a more complex neural network than our RLF. The neural network of ILF consists of 650,367 parameters, which are much larger than our RLF with 238,778 parameters. Consequently, for each smart contract in dataset **D1**, RLF generates an average of 290k transactions for testing but ILF only generates an average of 160k transactions. In addition, ILF uses imitation learning to learn from the symbolic executor, which may fail to generate enough effective test cases for fully training the neural network and limit the performance of ILF. However, RLF can be fully trained via fuzzing since fuzzing can efficiently generate a massive number of test cases so that we can select useful test cases for training. As a result, RLF detects more vulnerabilities than ILF with the same amount of time. As for Smartian, it

Table 8: Vulnerabilities reported by different tools in D2. TP is the number of True Positives and # is the number of reported vulnerabilities.

Bug ID	SmarTest [31]		Smartian [11]		ILF [15]		Our RLF	
	TP	#	TP	#	TP	#	TP	#
EL	46	46	19	19	15	15	20	20
SC	8	8	6	6	45	45	47	47
BD	/	/	92	98	90	92	90	92
UE	/	/	16	17	16	17	16	17
DD	/	/	0	0	1	1	1	1
EF	/	/	0	0	1	1	1	1

detects the fewest vulnerabilities than other tools though having no false positives. In general, our RLF outperforms state-of-the-art fuzzers in terms of efficiency in detecting vulnerabilities.

False Positives Analysis. We manually identify true vulnerabilities according to the labels in dataset **D1** and analyze the false positives (FP) reported by each tool. All FP results belong to Ether-Leaking vulnerabilities. We further observe that the three FP results reported by Mythril are from the same smart contract. These three FP functions are essentially safe since they are protected by a whitelist. We also note that SmarTest reported one FP, which is only *virtually* safe though predefined safety conditions can be violated. In addition, both RLF and ILF report two identical FP results, which are also virtually safe while satisfying the predefined test oracles. Totally, only a few false positives being reported by each tool and our RLF achieves 98.6% (i.e., $\frac{141}{143}$) precision rate in **D1**.

Answer to RQ2: RLF is more efficient in detecting Ether-Leaking (EL) and Suicidal (SC) vulnerabilities than other state-of-the-art tools. Within 30 minutes, RLF detects 8%-69% more vulnerabilities than the compared tools in **D1**. In addition, RLF achieves 98.6% precision rate in **D1**.

5.5 RQ3: Performance on Real-world Contracts

In this experiment, we evaluate the performance of vulnerability detection of RLF on real-world smart contracts. Due to the small proportion of vulnerabilities in real-world smart contracts, it is difficult to train a robust vulnerability-guided model (i.e., the neural network). To address this issue, we first train the model on dataset **D1** and then evaluate its performance on dataset **D2**. We run our RLF, ILF, Smartian and SmarTest for 1 minute per contract on dataset **D2**. The experimental results are reported in Table 8. Since the model of RLF is only trained for Ether-Leaking and Suicidal vulnerabilities, we divide the results into two parts for discussion: 1) the results on Ether-Leaking and Suicidal vulnerabilities and 2) the results on other vulnerabilities.

Ether-Leaking and Suicidal. Comparing RLF to coverage-guided fuzzers (i.e., Smartian and ILF), RLF generates vulnerable transactions more efficiently to reveal Ether-Leaking and Suicidal vulnerabilities. This result also demonstrates the effectiveness of our RLF on real-world smart contracts. In particular, RLF also detects more Suicidal (SC) vulnerabilities than SmarTest though it finds fewer Ether-Leaking (EL) vulnerabilities than SmarTest (i.e., 20 vs. 46). Through manually checking, we find that most of the extra vulnerabilities detected by SmarTest have execution paths with specific conditions, which need specific inputs of function to satisfy. Fig. 6 gives an example of this case, in which `msg.sender` can only transfer the ethers when `msg.value == 1 ether` and `_number == 1`. It is worth mentioning that it is very difficult for a fuzzer to generate these specific inputs to meet this condition (e.g., the probability of generating `_number == 1` for a random fuzzer is $1/2^{256}$). Thus, the above three fuzzers (i.e., RLF, Smartian and ILF) fail to reveal these vulnerabilities, while the symbolic executor of SmarTest can easily achieve this goal.

```
function play(uint256 _number){

    if(msg.value == 1 ether && _number <= 1)
        if (_number == 1)
        {
            // Ether-Leaking
            msg.sender.transfer(this.balance);
        }
    ...
}
```

Figure 6: Example of the vulnerability hiding in specific conditions.

Other Vulnerabilities. Although the model of RLF is trained for Ether-Leaking and Suicidal vulnerabilities, the vulnerable transaction sequences generated by RLF can still partly cover other vulnerabilities. For example, the vulnerable functions with Block State Dependence (BD) also have *Call* operations as the functions with Ether-Leaking vulnerability do; this feature enables RLF to cover Block State Dependence vulnerability while detecting Ether-Leaking vulnerability. In general, our RLF obtains satisfactory results in detecting these vulnerabilities, performing only a little worse than Smartian. Therefore, we believe that RLF can reveal more vulnerabilities through training on the dataset with specific vulnerabilities.

Answer to RQ3: Being trained on dataset **D1**, RLF outperforms other fuzzers in detecting Ether-Leaking (EL) and Suicidal (SC) vulnerabilities in real-world smart contracts in dataset **D2**. RLF also achieves close performance on other vulnerabilities compared to other fuzzers.

6 DISCUSSIONS

This section discusses RLF's limitations and potential future improvement.

6.1 Dependency on Vulnerability

Different from other coverage-oriented fuzzers, which mainly pursue high code coverage, we originally propose a vulnerability-guided fuzzer with a focus on detecting vulnerabilities. Experimental results show that our RLF outperforms other compared fuzzers. However, the performance of RLF heavily depends on the number of vulnerabilities for training. In our experiments, RLF receives the reward of vulnerability when test oracles identify any type of vulnerability. It is better to set the individual reward for each type of vulnerability so that RLF can learn to generate targeted vulnerable transaction sequences for different types of vulnerability. However, the number of uncommon vulnerabilities is not enough for training (e.g., Dangerous Delegatecall). Thus, we merge the reward of each vulnerability (i.e., R_{bugs}) as the reward of RLF.

6.2 Effectiveness of Actions

In our RLF, we adopt heuristic ideas to group functions with similar status operations so as to simplify the function space and construct our action space. In this way, the agent can partially handle the status operations by selecting the corresponding action (i.e., function groups) to generate the specific function-call sequences to trigger vulnerabilities. As shown in § 5.4, this action space is effective for most smart contracts. However, with the increased number of functions in smart contracts, each action (i.e., a function group) may contain more functions so the effectiveness of the action space would decline. To address this problem, we need a more precise method to group the functions for a better action space, which can easily upgrade our reinforcement learning framework.

6.3 Function Argument Selection

In our work, RLF only chooses functions to be invoked at each step, where the arguments of the function are randomly selected from seed pools. The random arguments may not satisfy the conditions in the tested function (e.g., statement `require`). Although we can repeatedly input different arguments to satisfy the conditions in functions, an appropriate selection of arguments may effectively improve the performance of fuzzing. In order to achieve this goal, it is necessary to design a revised neural network architecture for handling the selection of both functions and arguments. However, this revised neural network (e.g., the network of ILF) may further complicate the entire system design and also bring extra overheads, thereby significantly slowing down the testing process. Therefore, to handle this performance trade-off, we should adopt appropriate strategies for different fuzzing scenarios.

7 RELATED WORK

Symbolic execution and fuzzing are the mainstream techniques that support generating transactions to explore the execution paths of smart contracts and reveal their vulnerabilities.

Symbolic Execution. The core idea of symbolic execution [18] is to execute the program with symbolic input rather than concrete input. With the symbol input, the symbolic executor systematically explores all the possible execution paths and obtains their constraints. The constraint solvers then calculate the concrete instances of each constraint, which can be used as input to trigger the corresponding execution path.

Symbolic execution has been commonly used to generate transactions to test smart contracts. In 2016, Luu et al. [20] proposed a novel symbolic executor, namely Oyente, which detects four classic vulnerabilities in smart contracts. In addition to Oyente, other symbolic executors such as Mythril [25], Manticore [24], Maian [27], Verx [28], Mpro [36] have been proposed to detect vulnerabilities in smart contracts. Moreover, recent studies proposed vulnerability-guided symbolic executors to improve the efficiency of vulnerability detection. In particular, Krupp et al. [19] presented a directed symbolic executor, namely TEETHER, through analyzing the critical paths with key instructions (e.g., ether transfer or code injection). So et al. [31] presented SMARTEST to exploit language model-guide symbolic execution to obtain the vulnerable transaction sequences. Compared to these previous approaches, the proposed RLF comprehensively considers both the code coverage and vulnerability so as to identify the vulnerabilities across multiple functions.

Fuzzing. Fuzzing has been increasingly adopted for fuzzing smart contracts. Jiang et al. [17] presented ContractFuzzer, which generates fuzzing inputs based on the ABI specifications of smart contracts and defines test oracles to detect security vulnerabilities. Based on AFL [35], sFuzz [26] designs an efficient lightweight multi-objective adaptive strategy with consideration of branch distance. Harvey [34] is a commercial (closed-source) fuzzer that employs a heuristic for predicting new inputs so that it is more likely to cover new paths. The above fuzzers mainly focus on generating specific inputs to satisfy complex conditions while failing to consider the transaction sequences to trigger vulnerabilities.

To generate the transaction sequences, ILF [15] uses imitation learning to learn from symbolic execution experts for generating effective transaction sequences. In addition, SMARTIAN [11] employs both static and dynamic analyses for fuzzing smart contracts. However, both of them are coverage-guided fuzzers that may waste a lot of time exploring the non-vulnerable transaction sequences. By contrast, as a vulnerability-guided fuzzer, our RLF can reduce the time for generating vulnerable transaction sequences.

In addition, it is worth noting that *machine learning* has been commonly used in fuzzing outside the field of smart contracts and has achieved outstanding results [8, 13, 30, 39].

8 CONCLUSION

In this work, we present a vulnerability-guided fuzzer based on reinforcement learning, namely RLF, for effectively generating vulnerable transaction sequences in smart contracts. In particular, we firstly model the process of fuzzing smart contracts as MDP to construct our reinforcement learning framework. Then, we design a new appropriate reward with consideration of both vulnerability and code coverage so as to guide RLF to generate specific transaction sequences for revealing vulnerabilities, especially for the complicated vulnerabilities related to multiple functions. Finally, we design a neural network for RLF to automatically learn from the previous sequences, so that our RLF can effectively generate vulnerable transaction sequences. Extensive experimental results demonstrate that the proposed RLF detects more vulnerabilities than other state-of-the-art tools within a limited time. Our results also validate the effectiveness of the designed reward (considering both vulnerability and code coverage) contributing to the superior performance of RLF.

ACKNOWLEDGMENTS

The research has been supported by the National Key R&D Program of China (2020YFB1006002), Technology Program of Guangzhou, China (202103050004), Hong Kong RGC Projects (PolyU15219319, PolyU15222320, PolyU15224121), and HKBU COMP Department Start-up Fund (41.4541.179432)

REFERENCES

[1] 2019. ILF. https://github.com/eth-sri/ilf

[2] 2020. VeriSmart-benchmarks. https://github.com/kupl/VeriSmart-benchmarks

[3] 2022. ERC-721 NON-FUNGIBLE TOKEN STANDARD. https://ethereum.org/en/developers/docs/standards/tokens/erc-721/

[4] 2022. Pytorch. https://pytorch.org/

[5] 2022. Solidity Documentation. https://docs.soliditylang.org/en/v0.8.16/

[6] 2022. Xblock. http://xblock.pro

[7] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1621–1633. https://doi.org/10.1145/3510003.3510230

[8] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep Reinforcement Fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*. 116–122. https://doi.org/10.1109/SPW.2018.00026

[9] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining Smart Contract Defects on Ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2022), 327–345. https://doi.org/10.1109/TSE.2020.2989002

[10] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2021. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2021), 1433–1448. https://doi.org/10.1109/TETC.2020.2979019

[11] Jaeseung Choi, Gustavo Grieco, Doyeon Kim, Alex Groce, Soomin Kim, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *The 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE/ACM*. 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[12] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 530–541. https://doi.org/10.1145/3377811.3380364

[13] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 50–59. https://doi.org/10.1109/ASE.2017.8115618

[14] Matthew Hausknecht and Peter Stone. 2015. Deep Recurrent Q-Learning for Partially Observable MDPs. In *AAAI 2015 Fall Symposium*.

[15] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. https://doi.org/10.1145/3319535.3363230

[16] Huawei Huang, Wei Kong, Sicong Zhou, Zibin Zheng, and Song Guo. 2021. A survey of state-of-the-art on blockchains: Theories, modelings, and tools. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–42.

[17] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[18] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[19] Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1317–1333.

[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[21] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[22] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. (2013), 3111–3119.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[24] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[25] Bernhard Mueller. 2018. Smashing Ethereum Smart Contracts for Fun and Real Profit. In *The 9th Annual HITB Security Conference in The Netherlands (HITB SECCONF), Amsterdam*, Vol. 9. 54.

[26] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[27] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[28] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1661–1677. https://doi.org/10.1109/SP40000.2020.00024

[29] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. 2019. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605* (2019).

[30] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. 2019. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133* (2019).

[31] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.

[32] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. 67–82.

[33] Friedhelm Victor and Andrea Marie Weintraud. 2021. Detecting and Quantifying Wash Trading on Decentralized Cryptocurrency Exchanges. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/3442381.3449824

[34] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.

[35] Michal Zalewski. 2014. American fuzzy lop (2017). *URL http://lcamtuf. coredump. cx/afl* 14 (2014), 28.

[36] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven Stewart, and Vijay Ganesh. 2019. MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 456–462. https://doi.org/10.1109/ISSRE.2019.00052

[37] Peilin Zheng, Zibin Zheng, Jiajing Wu, and Hong-Ning Dai. 2020. XBlock-ETH: Extracting and Exploring Blockchain Data From Ethereum. *IEEE Open Journal of the Computer Society* 1 (2020), 95–106. https://doi.org/10.1109/OJCS.2020.2990458

[38] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.

[39] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-Box Fuzzing through Deep Learning. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 127, 15 pages.