Smart Contract Code Repair Recommendation based on Reinforcement Learning and Multi-metric Optimization

HANYANG GUO, School of Software Engineering, Sun Yat-Sen University, China
YINGYE CHEN, School of Computer Science and Engineering, Sun Yat-Sen University, China
XIANGPING CHEN, School of Communication and Design, Sun Yat-Sen University, China
YUAN HUANG and ZIBIN ZHENG, School of Software Engineering, Sun Yat-Sen University, China

A smart contract is a kind of code deployed on the blockchain that executes automatically once an event triggers a clause in the contract. Since smart contracts involve businesses such as asset transfer, they are more vulnerable to attacks, so it is crucial to ensure the security of smart contracts. Because a smart contract cannot be tampered with once deployed on the blockchain, for smart contract developers, it is necessary to fix vulnerabilities before deployment. Compared with many vulnerability detection tools for smart contracts, the amount of automatic fix approaches for smart contracts is relatively limited. These approaches mainly use defined pattern-based methods or heuristic search algorithms for vulnerability repairs. In this article, we propose *RLRep*, a reinforcement learning-based approach to provide smart contract repair recommendations for smart contract developers automatically. This approach adopts an agent to provide repair action suggestions based on the vulnerable smart contract without any supervision, which can solve the problem of missing labeled data in machine learning-based repair methods. We evaluate our approach on a dataset containing 853 smart contract programs (programming language: Solidity) with different kinds of vulnerabilities. We split them into training and test sets. The result shows that our approach can provide 54.97% correct repair recommendations for smart contracts.

CCS Concepts: • **Software and its engineering** → *Automatic programming*; • **Security and privacy** → **Software security engineering;**

Additional Key Words and Phrases: Repair recommendation, smart contract

ACM Reference format:
Hanyang Guo, Yingye Chen, Xiangping Chen, Yuan Huang, and Zibin Zheng. 2024. Smart Contract Code Repair Recommendation based on Reinforcement Learning and Multi-metric Optimization. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 106 (April 2024), 31 pages.
https://doi.org/10.1145/3637229

The research is supported by the National Natural Science Foundation of China (62032025, 61976061), the Key-Area Research and Development Program of Shandong Province (2021CXGC010108), and Guangdong Basic and Applied Basic Research Foundation (2023A1515010746).

Authors' addresses: H. Guo, Y. Huang, and Z. Zheng (Corresponding author), School of Software Engineering, Sun Yat-Sen University, Zhuhai, Guangdong, China, 519000; e-mail: guohy36@mail2.sysu.edu.cn; Y. Chen, School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou, China; X. Chen, School of Communication and Design, Sun Yat-Sen University, Guangzhou, China; e-mail: chenxp8@mail.sysu.edu.cn.

1049-331X/2024/04-ART106 $15.00
https://doi.org/10.1145/3637229

1 INTRODUCTION

The smart contract is a piece of code running in the blockchain network [21, 37]. It realizes the automatic processing of traditional contracts in the form of computer instructions and completes the business logic given by the user [82]. Usually, smart contracts are developed using a high-level programming language (e.g., Solidity) and then compiled into bytecode to deploy on the blockchain platform (e.g., Ethereum) [49]. Due to the immutability of blockchain data, once the smart contract is deployed on the blockchain, it cannot be modified. Therefore, it is critical to keep the smart contract as vulnerability-free as possible before deployment. Unfortunately, many smart contracts deployed on the blockchain still have vulnerabilities inevitably. Since many asset transfers are involved in smart contracts, these vulnerabilities are easily exploited by attackers and cause serious economic losses. Table 1 illustrates several famous smart contract security events. These events were caused by different vulnerabilities in smart contracts and caused millions of ethers (ETH, the digital token in Ethereum) and dollars to be stolen.

To ensure the security of smart contracts, many researchers have proposed various smart contract vulnerability detection methods to detect whether there are vulnerabilities in smart contracts. These approaches adopt static or dynamic analysis methods such as symbolic execution [32, 44, 47, 60] or fuzzing [20, 23, 40] to detect security vulnerabilities in smart contracts. Developers can modify these vulnerabilities manually during the contract audit stage to avoid deploying insecure contracts to the blockchain. Besides, some development guidance documents [11, 45] are proposed to provide suggestions for developers to write smart contracts, thereby helping developers to write secure smart contracts.

Nonetheless, the security issue of smart contracts on Ethereum is still unresolved. The vulnerable contracts still threaten the ecology of the Ethereum blockchain. On the one hand, although there are many guidance documents about smart contract development, many smart contracts on the blockchain are not written following the development specifications. This leads to a large number of on-chain contracts deployed with security vulnerabilities every day [77]. On the other hand, existing smart contract vulnerability detection methods can only determine the type of vulnerability and its location, but do not provide correct fixes and recommendations [77]. The process of fixing the contract still needs to be done manually. Manually fixing smart contract codes is time-consuming and labor-intensive. Research [6] shows that developers spend about half of all development time on software fixes. Therefore, it is necessary to help developers quickly repair smart contracts by automatically repairing or providing repair suggestions. In traditional programming languages (e.g., Java, C++), automatic repair approaches use different techniques [16] such as heuristic search [18, 35, 36], defined repair patterns [3, 24, 25], machine learning [26, 62, 71, 81], and so on. Compared with other two kinds of techniques, machine learning-based methods can fix more kinds of vulnerabilities [26]. As for the programming language of the smart contract (e.g., Solidity[1]), there are some repair approaches and fix recommenders that are based on heuristic search methods [75] or defined repair patterns [77]. These approaches have limited effectiveness in fixing vulnerabilities, and some types of vulnerabilities cannot be fixed well, since the rules are difficult to design and define. For example, the results shown in Reference [75] indicate that the heuristics method cannot fix Integer Overflow vulnerabilities well. But few approaches adopt machine learning-based techniques. It is because the current mainstream machine learning-based program repair methods use the **Neural Machine Translation (NMT)** framework [74, 81], which is a supervised learning method that requires a large number of labeled data (that is, programs with vulnerabilities and corresponding successfully repaired programs). But, since Solidity is an emerging programming language, such datasets are lacking. Therefore, it is difficult to adopt machine

[1]In this article, we only focus on Solidity, as it is the most popular development language for smart contracts

Table 1. Famous Smart Contract Security Events

Name	Date	Ether Loss	USD loss
DAO [78]	June 17, 2016	3,600,000	79.6 million
Parity [8]	Nov. 6, 2017	513,701	153 million
bZx [66]	Feb. 15 & Feb. 18, 2020	3,581	0.942 million
Uniswap [13]	Apr. 18, 2020	1,278	0.22 million
Poly Network [1]	Aug. 10, 2022	55,000	610 million
Akutar [14]	Apr. 23, 2022	11,539	34 million

learning-based program repair methods or fix recommenders in Solidity. Besides, although there are some approaches adding perturbations or using broker models to generate vulnerable codes [72, 73], these vulnerable codes do not come from the real world and may not extrapolate well to the real distribution. Additional time is also consumed to train the broken model.

To address these issues, we propose a Solidity fix recommender based on reinforcement learning called *RLRep*. *RLRep* utilizes an agent to imitate the actions of humans to fix vulnerabilities in smart contracts. In contrast to machine learning-based repair methods in the traditional programming language, *RLRep* does not require a large amount of data with labels. It only performs expert demonstration pre-training [2] with a small amount of labeled data. Besides, most of the data are collected from real smart contracts deployed in Ethereum rather than generated by tools. During formal training, it only inputs relevant information about vulnerable programs and obtains correct repair action suggestions in the form of self-exploration rather than through supervision or prior knowledge. Specifically, we input the vulnerable smart contract source code and its corresponding AST sequence information into the model. Then, we utilize an encoder-decoder model to generate repair action paths. The design of repair action is based on official fix suggestion and mutation operators of traditional programming language [22, 52, 58, 76] and Solidity [38, 68]. Subsequently, we construct a reward function not only based on quality performance but also based on the exclusive vulnerability feature of smart contracts. Specifically, it is based on the compilation, vulnerability detection tools [12, 44, 47, 60], code similarity [15], and code entropy [56, 61]. The results of the reward function guide the agent to recommend the correct repair action path to achieve multi-metric optimization. To train the fix recommender, we build 853 smart contracts with vulnerabilities by collecting from Etherscan[2] (a blockchain explorer for the Ethereum network) and using vulnerability injection tools. These smart contracts include five types of typical vulnerabilities in smart contracts, that is, **exception disorder (ED), integer overflow (IO), reentrancy (RE), transaction order dependence (TOD)**, and **tx.origin (TX)**. We split the dataset into the training set and test set. The evaluation result shows that *RLRep* can provide 54.97% correct repair suggestions, which is better than the baseline approaches. Besides, we also adopt test cases to verify the reliability [43, 79] of fixed smart contracts. To facilitate research and application, we make *RLRep* and the dataset permanently available on GitHub.[3]

The contributions of this article are shown as follows:

(1) We propose a reinforcement learning approach to build a program repair recommender in the field of smart contracts. It utilizes policy gradient algorithm to guide the model to generate fix suggestions without supervision.
(2) To solve the problem of insufficient labeled datasets, we adopt a small amount of labeled data for pre-training and then collect and construct unlabeled data. Most of the collected

[2] http://www.etherscan.io/
[3] https://github.com/Anonymous123xx/RLRep

```
1  contract Lotto {
2   bool public payedOut = false;
3   address public winner;
4   uint public winAmount;
5   ...
6   function sendToWinner() public {
7     require(!payedOut);
8 -    winner.send(winAmount);
9 +    require(winner.send(winAmount);
10    payedOut = true;
11  }
12  ...
13 }
14
```

Fig. 1. Exception disorder.

data come from real smart contracts deployed in Ethereum. Only a few training samples are generated by the vulnerability injection tool. Based on the vulnerable code data, a self-exploration method is used to train a reinforcement learning-based repair recommendation model.

(3) We design a list of repair actions to guide the agent to generate repair action paths. The actions are designed based on official fix suggestion and mutation operators of traditional programming language and Solidity.

(4) We design the reward function based on compilation, vulnerability detection tools, code similarity, and code entropy, which can achieve optimization of smart contract repair recommendation from multi metrics. And our method outperforms existing state-of-the-art methods based on heuristic search.

2 BACKGROUND AND OVERVIEW

2.1 Typical Vulnerabilities in Insecure Smart Contract

As mentioned before, there are still many insecure smart contracts that contain many common patterns. Numerous studies and related technical documents (e.g., Smart Contract Weakness Classification Registry[4]) have classified different vulnerability types according to these patterns, and recommend related fix suggestions [17, 30]. In this article, we will focus on five types of vulnerabilities and describe how to fix them. The five types of vulnerabilities are **exception disorder (ED), integer overflow (IO), reentrancy (RE), transaction order dependence (TOD)**, and **tx.origin (TX)**.

Exception Disorder means that (1) the smart contract does not check the return value of *send* and *call*, resulting in inconsistent accounts between the two parties, or (2) the return value of the *call* is not checked, resulting in an exception that cannot be passed up and a logic error occurs. *send* and *call* are the APIs used to implement ETH transfers, so their use is closely related to the account's security. Figure 1 illustrates an example of this type of vulnerability. The transfer function *send* is used in line 8, but its return value is not judged, so even if the *send* function fails to execute, the state modification in line 10 will still be executed, which is obviously wrong. The fix suggestion of this kind of vulnerability is to replace the low-level function (e.g., *send*, *call*) with the high-level function (e.g., *transfer*) to achieve transfer operation. As for this example, it is executed in line 8. Another fix suggestion is to insert *require* statement to handle exceptions. As for this example, line 8 can be replaced with *require(winner.send(winAmount));* [77]. The purpose of the *require* statement is to allow *send* execution to throw an exception and revert if it fails.

[4] https://swcregistry.io/

```
1  contract IntOverflow{
2    function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
3      uint256 amount = uint256(cnt) * _value;
4  +   assert(amount / uint256(cnt) == _value);
5      require(cnt > 0 && cnt <= 20);
6      require(_value > 0 && balances[msg.sender] >= amount);
7      balances[msg.sender] = balances[msg.sender].sub(amount);
8      for (uint i = 0; i < cnt; i++){
9        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
10       Transfer(msg.sender, _receivers[i], _value);
11     }
12     return true;
13   }
14 }
15
```

Fig. 2. Integer overflow.

```
1  contract Reentrancy {
2    mapping(address => uint) public balance;
3    function withdraw(uint _amount) {
4      if(balance[msg.sender] >= _amount) {
5  +     balance[msg.sender] -= _amount;
6        msg.sender.call.value(_amount)();
7  -     balance[msg.sender] -= _amount;
8      }
9    }
10   function() public payable {}
11 }
12 ------------------------------------------------
13 ------------------------------------------------
14 contract Attack {
15   Reentrance public entrance;
16   constructor(address _target) public {
17     entrance = Reentrance(_target):
18   }
19   function attack() payable {
20     entrance.withdraw(0.5 ether);
21   }
22   function() public payable {
23     entrance.withdraw(0.5 ether);
24   }
25 }
26
```

Fig. 3. Reentrancy.

Integer Overflow exists in different programming languages. Integer types in programming languages have a range, that is, an integer type has a maximum value and a minimum value. Integer overflow or underflow occurs when an out-of-bounds occurs in an arithmetic operation. The data bits of uint256 in the **Ethereum Virtual Machine (EVM)** is 0~255. When the value exceeds this range, overflow will occur. Therefore, after the operation of "+", "-", "*", and "/" This vulnerability occurs when the variable is not validated in time for its value. Figure 2 illustrates a case. There is a multiplication operation in line 3, and the *amount* lacks an overflow judgment, so there is a possibility of overflow [29]. The fix suggestions for this kind of vulnerability are adding validation conditions to all integer variables in the previous line or the next line of the vulnerable line. Another suggestion is using the *add*, *sub*, *multi*, and *div* functions in the *SafeMath* library for calculation [77].

Reentrancy means that the attacked smart contract is abnormally re-entered (called again), resulting in inconsistent results. From the perspective of contract auditing, it can be assumed that all external calls of the contract may be subject to reentrancy attacks. Figure 3 illustrates a case. When the attacker calls the *attack* function in the contract *Attack* (line 19), the function will execute

```
1  contract TransactionOrderDependence {
2    event Transfer(address indexed from, address indexed to, uint256 value );
3    event Approval(address indexed owner, address indexed spender, uint256 value);
4    mapping(address => uint256) private _balances;
5    mapping(address => mapping (address => uint256)) private _allowed;
6    uint256 private _totalSupply;
7    constructor(uint totalSupply) {
8      _balances[msg.sender] = totalSupply;
9    }
10   ...
11   function approve(address spender, uint256 value) public returns (bool) {
12     require(spender != address(0));
13+    require(_allowed[msg.sender][spender] == 0 || value == 0);
14     _allowed[msg.sender][spender] = value;
15     emit Approval(msg.sender, spender, value);
16     return true;
17   }
18   ...
19 }
20
```

Fig. 4. Transaction order dependence.

the *withdraw* function in the contract *Reentrance* through line 20. When the contract *Reentrrance* executes the *withdraw* function (line 3), it will send ether to the *Attack* contract (line 5) using a call statement. However, in the Solidity language, when an external account or other contract sends ether to a contract address, the callee contract's fallback function will be executed. So, the contract *Attack* responds to the transfer using the Attack.fallback function (line 22). The *Attack.fallback* function calls the *Reentrance.withdraw* function to extract the ether again (line 23). Therefore, the contract *Attack* will always withdraw ether from the contract *Reentrance* until the gas (Energy consumed when executing smart contracts) is exhausted, and the deduct-statement (line 7) that deducts the number of tokens held by the contract Attack is executed only once. The suggestion to fix this kind of vulnerability is to update the state variable (account balance) before executing the external call code line [77]. Another suggestion is replacing *call.value* function to *transfer* function or *send* function, because the latter two functions will limit the gas usage (up to 2,300 gas) [7]. For example, in Figure 3, this suggestion can be adopted in line 6.

Transaction Order Dependence means that it takes a certain amount of time for a transaction to be propagated and approved by miners to be included in a block. If an attacker monitors a target transaction of the corresponding contract in the network, and then sends his own transaction to change the current contract status such as a transaction reducing the contract payoff in the bounty contract, then it has a certain probability that these two transactions are included in the same block. As a result, the attacker's transaction can queue before the other transaction to complete the attack. Figure 4 shows an example of a contract based on the ERC20 standard. The ERC20 standard is quite well known for building tokens on Ethereum. This standard has a potential transaction order dependence vulnerability that arises due to the *approve* function. As it is shown in Figure 4, by executing *Approval* method in line 14, a user A (i.e., *msg.sender*) allows a user B (i.e., *spender*) to transfer N tokens by calling *approval* function. If A wants to change the allowed transfer amount from N to M, but B has already sent a transaction to transfer N and executed it first, then B can transfer N tokens before A modifies the transfer amount, resulting in A's additional loss. One of the fix suggestions is to add a field to the inputs of *approve*, which is the expected current value and to have approved revert if B's current allowance is not what A indicated she was expecting [65]. For example, a statement *require(_allowed [msg.sender][spender]==0||value==0);* can be added before line 13.

Tx.origin is a global variable in Solidity. It can traverse the call stack and return the address of the account that originally sends the call. In other words, tx.origin is the address of the

```
1  contract TxOrigin {
2    address owner;
3    function TxOrigin() public {
4      owner = msg.sender;
5    }
6    function sendTo(address receiver, uint amount) public {
7  -   require(tx.origin == owner);
8  +   require(msg.sender == owner);
9      receiver.transfer(amount);
10   }
11
12 }
13
```

Fig. 5. Tx.Origin.

contract that was originally in the entire transaction process. Using authentication that adopting the tx.origin variable in a smart contract can cause the contract to pass the authorization check and to be attacked. Figure 5 shows an example. Line 7 will cause the contract to be attacked. The fix suggestion is to replace *tx.origin* to *msg.sender* during authorization check, which can be executed in line 7 [7].

Discovery. It can be found that the fix suggestions for the above vulnerabilities operate on the vulnerable line or the previous and next line of the vulnerable line, so this scope can be used as the model input that should be concerned.

2.2 Encoder-decoder Model and Reinforcement Learning

In this article, we combine the encoder-decoder model [4] and reinforcement learning [46] to recommend repair actions. The encoder-decoder model is a kind of text generation framework that is utilized in the field of machine translation, text simplification, text summarization, question answering system, and so on. Its input and output are two kinds of text sequences. We use **Long Short-Term Memory (LSTM)** network to encode and decode the sequences because of the good performance in code repair [50, 74, 81]. The attention mechanism is also utilized to solve the problem of information loss caused by the long sequence [51].

Reinforcement learning refers to the agent learning the policy in a "trial and error" way, and the reward guides the behavior obtained by interacting with the environment. The goal is to make the agent obtain the optimal policy with the greatest reward. Reinforcement learning is different from supervised learning mainly in the reinforcement signal. The reinforcement signal provided by the environment in reinforcement learning is an evaluation (usually a scalar signal) of the quality of the generated action, rather than telling the reinforcement learning system how to generate correct action (i.e., labeled data).

In this article, we take the code sequence and AST sequence as input text sequences and the fix operators as the output sequence in the encoder-decoder model. Fix suggestions for smart contracts can be obtained by executing the sequence of operators. We adopt policy gradient, a kind of reinforcement learning algorithm to get the optimal fix operator sequences for the recommendation. The detailed process is presented in the next section.

3 METHODOLOGY

In this section, we present the whole process of the smart contract repair recommendation and the proposed repair recommendation approach, and *RLRep* is also introduced in detail. As shown in Figure 6, we first extract smart contracts from Etherscan and utilize vulnerability detection tools (e.g., *Mythril* [47], *Securify* [60], *Oyente* [44], *Slither* [12]) to determine the vulnerability type and localize the vulnerable function. Then, we determine the vulnerable line in the vulnerable function

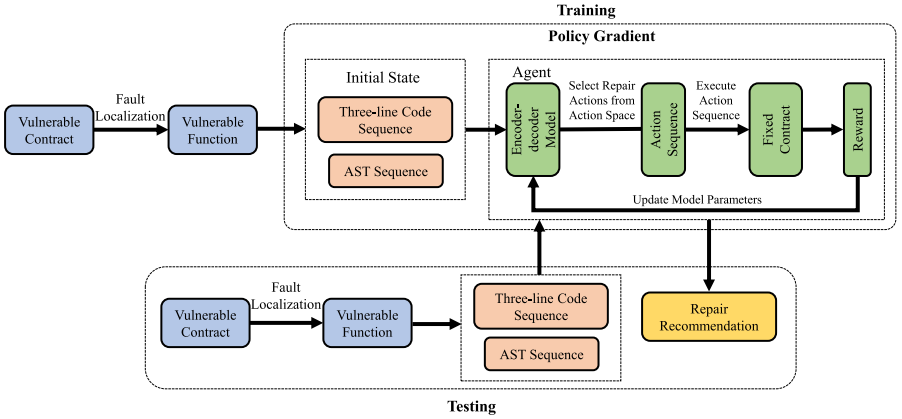

Fig. 6. The overview of *RLRep*.

and extract the previous and the next line of the vulnerable code line as the context to construct the three-line source code sequence. Besides, to get the syntax and structure information of the vulnerable function, we also parse and traverse the **abstract syntax tree (AST)** as one of the inputs. After that, we adopt the policy gradient algorithm. We input the source code sequence and AST sequence into an encoder-decoder model and output the repair action sequence selected from the action space. The action space is the set of all repair actions. The design of repair actions is based on two parts. One is the official fix suggestions proposed in Section 2.1. The other is mutation operators based on well-known mutants of traditional programming languages [22, 52, 58, 76] and Solidity language features [38, 68]. By executing the repair actions on the vulnerable contract, we can obtain a fixed contract. We design a reward function based on compilation, vulnerability detection tools, code similarity, and code entropy to evaluate the fixed contract. We back-propagate the evaluation results to update the model parameters and adjust the probability of action selection to get the maximum reward. In the testing phase, we also extract the vulnerable function from the vulnerable contract and input the three-line source code sequence and AST sequence to get the repair recommendations. In particular, we adopt beam search to provide several repair recommendations. Developers can fix vulnerable codes more efficiently based on repair recommendations [80].

3.1 Data Collection and Processing

RLRep provides corresponding fix suggestions based on the source code of smart contracts, so it is necessary to collect the source code dataset of smart contracts. Smart contracts deployed on Ethereum are not open source and only contain corresponding bytecode information. Therefore, we collect the source data of smart contracts from Etherscan. Etherscan is a blockchain browser that enables viewing public data related to transactions, smart contracts, addresses, and other content on the Ethereum blockchain, including the source code of open-source smart contracts. To collect the vulnerable smart contracts, we utilize several vulnerability detection tools to extract smart contracts with different vulnerabilities. According to the performance analysis of different tools in detecting different types of vulnerabilities shown in Reference [17], we select the best detection tool for each type of vulnerability, that is, *Mythril* [47], *Securify* [60], *Oyente* [44], and *Slither* [12]. Specifically, we utilize *Mythril* to detect ED and RE vulnerabilities. *Mythril* is an officially recommended smart contract security analysis tool for Ethereum that uses symbolic execution to detect various security vulnerabilities in smart contracts. *Securify* is utilized to detect TOD vulnerabilities. It is also the tool being used for security vulnerability analysis of Ethereum smart contracts based

on symbolic execution. *Oyente* is used to detect IO vulnerabilities. It is used to detect potential security vulnerabilities in contract code and is a testing tool based on symbolic execution techniques. *Slither* is used to detect TX vulnerabilities. It is a static analysis framework for smart contracts that adopts data flow analysis techniques. The input of these tools is the smart contract source code. In addition, considering the accuracy of detection tools, we also manually checked and filtered out false positive contracts. In particular, as for RE, since it is classified as benign and malicious, our manual check also filtered out benign RE and retained the RE that possibly makes the contract vulnerable (i.e., malicious RE). The detailed manual check process is shown as follows: As for each vulnerability type, we invite two volunteer students who have two years of Solidity development experience to verify whether each positive detection result is correct. The verification refers to the smart contract vulnerability library **Smart Contract Weakness Classification (SWC)** Registry[5] and the characterization of the specific vulnerability type in Solidity's official documentation.[6] Ten postgraduate students in total are invited to participate in this work. If both students agree that the detection tool's result is correct, then the contract is considered vulnerable. If both students agree that the results of the detection tool are incorrect, then the contract is considered a false positive. If the two students disagree, then the two parties finalize the result through discussion. Besides, to address the issue of the limited dataset, we adopt a vulnerability injection tool [17] to increase the number of smart contracts with ED vulnerabilities and TX vulnerabilities. Thereby, we build a dataset of vulnerable smart contracts.

As for each vulnerable smart contract, we localize the vulnerable function and the vulnerable code line with the use of detection tools. Because code changes of vulnerability fixing are diverse significantly due to different contexts [39, 69], we extract the vulnerable code line, the previous code line of the vulnerable code line, and the next code line of the vulnerable code line to build the three-line vulnerable code snippet. The reason why three lines of code are extracted is to ensure that the length of the code snippet is not too long to input the encoder-decoder model while obtaining the most related context information of the vulnerable line. After that, we traverse the three-line code snippet into the three-line code sequence. Subsequently, to obtain the structure, syntax, and type information of the code tokens, we utilize a Solidity parser built on ANTLR[7] to parse the AST of the vulnerable function. The reason why we obtain this information is that a study [41] found that repair actions are utilized differently with respect to different structures and syntax information. We convert the AST to sequence information by preorder traversal. In particular, because we need to train an encoder-decoder model, it is necessary to control the AST vocabulary size while retaining the syntax information. Therefore, we refer to a method for reducing vocabulary size proposed in Reference [62]. Specifically, by utilizing the Solidity parser, we can obtain the role of each AST identifier token (e.g., variable declaration, function definition, and so on) or its literal type (e.g., string, Boolean, hex number). In each vulnerable function, each unique identifier or literal is mapped to an ID. For example, a variable declaration AST token is denoted by ID *VariableDeclaration_#*. # is a numerical ID generated sequentially for each corresponding identifier or literal within a vulnerable function (e.g., the first function definition will receive FunctionDefinition_0, the third string value will receive StringLiteral_2). We count the 300 identifiers and literals with the largest number and keep them in the vocabulary, other identifiers, and literals except these are replaced by the above mapping IDs. To preserve the original semantic information of the three-line source code, we do not process the source code like AST. Thereby, we obtain the three-line source code sequence and the abstracted AST sequence.

[5] https://swcregistry.io/
[6] https://docs.soliditylang.org/
[7] https://github.com/antlr/antlr4

3.2 Encoder-decoder Model

The encoder-decoder model is shown in Figure 7. Two inputs are adopted in the encoder-decoder model. One is the three-line code sequence, the other is the abstracted AST sequence. Therefore, our encoder-decoder model consists of two encoders and one decoder. First, each sequence can be represented as $x = [x_1, x_2, \ldots, x_i, \ldots, x_L]^T$, where x_i represents each token and L is the sequence length (Padding mask module is adopted to ensure the identified sequence length [50]). The three-line code sequence and the AST sequence are input to a word embedding layer. The detailed process is shown as follows:

$$v_{st} = W_s x_t, \tag{1}$$

where W_s is the embedding matrix of code sequence tokens, x_t is the source code token at timestep t, v_{st} is the embedding vector of source code token at timestep t. Similarly, for the AST sequence, we utilize the word embedding layer to embed each token $x_t^{'}$ to embedding vector v_{at}.

$$v_{at} = W_a x_t^{'}, \tag{2}$$

where W_a is the embedding matrix of AST sequence tokens. The code token vectors and the AST token vectors constitute the code vector sequence and the AST vector sequence, respectively. We input these two kinds of vector sequences to obtain the hidden states h_t and $h_t^{'}$ at each timestep t.

$$h_t = lstm(h_{t-1}, v_s) \tag{3}$$

$$h_t^{'} = lstm\left(h_{t-1}^{'}, v_a\right) \tag{4}$$

We also adopt the attention mechanism to assign hidden state weight to improve the performance of encoder-decoder model. Since there are two sets of hidden states, we utilize two attention layers to get the context vector set, which is shown as follows:

$$c_{t'} = \sum_{t=1}^{T} \alpha_{t't} h_t + \sum_{t=1}^{T'} \alpha_{t't}^{'} h_t^{'}, \tag{5}$$

where $c_{t'}$ is the context vector at timestep t, T and $T^{'}$ are the lengths of the source code sequence and the AST token sequence. $\alpha_{t't}$ and $\alpha_{t't}^{'}$ are the hidden states' weight distribution of source code and AST, which is calculated as follows:

$$\alpha_{t't} = \frac{exp(e_{t't})}{\sum_{k=1}^{T} exp(e_{t'k})}, \tag{6}$$

$$\alpha_{t't}^{'} = \frac{exp(e_{t't}^{'})}{\sum_{k=1}^{T'} exp(e_{t'k}^{'})}, \tag{7}$$

where,

$$e_{t't} = a(s_{t'-1}, h_t), \tag{8}$$

$$e_{t't}^{'} = a(s_{t'-1}, h_t^{'}). \tag{9}$$

a is the alignment model aiming to evaluate the correlation between the input token at timestep t and the output token at timestep $t^{'}$.

In the decoder, we aim to generate repair action token ID $y_{t'}$ based on context vector $c_{t'}$ and previously generated repair action. The probability is shown as follows:

$$P(y_{t'}|y_1, y_2, \ldots, y_{t'-1}, c_{t'}) = p(y_{t'-1}, s_{t'}, c_{t'}), \tag{10}$$

where $s_{t'}$ is the hidden state. The goal of model training is to maximize the probability of correct repair action selection at each timestep and then generate a repair action sequence.

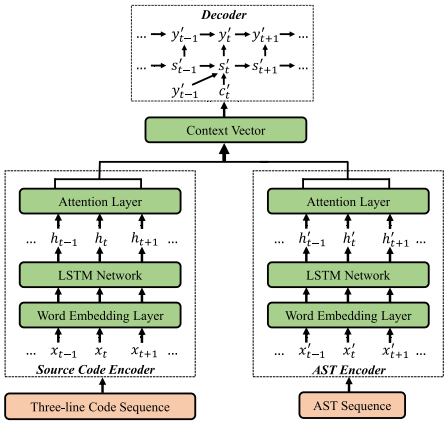

Fig. 7. Encoder-decoder model.

Besides, to expand the search space of the fix suggestions and improve the performance of *RLRep*, we use a heuristic algorithm called beam search [31]. Beam search is an algorithm used in the encoder-decoder model to optimize output results. It attempts to optimize the search space on a breadth-first basis (similar to pruning) for the purpose of reducing memory consumption. It is usually used in various NLP generation-like tasks, such as machine translation, dialogue systems, and text summarization. Compared with greedy search, beam search prefers to select several fix suggestions with the highest output probability rather than one fix suggestion. At each timestep, beam search selects k most likely repair actions, where k is the beam size. Then it will cut off other selections and go to the next timestep to select repair actions until meeting the end-of-sequence symbol. Therefore, k repair recommendations for each vulnerable smart contract can be provided.

3.3 Policy Gradient

In this article, we utilize policy gradient, one kind of reinforcement learning algorithm to generate repair recommendations. Policy gradient adopts the deep learning network to output the fix suggestions with the maximum reward according to the state input. We combine the encoder-decoder model with policy gradient. The execution process of the encoder-decoder model can be regarded as **Markov Decision Process (MDP)** [53] in a continuous state. As mentioned above, given source code texts x and AST sequence texts $x^{'}$, the encoder-decoder model selects a repair action $y_{t'}$ at each timestep $t^{'}$, thereby constitutes a repair sequence (i.e., the policy). Each repair action is obtained by selecting the action with the highest probability in the action space according to the conditional probability distribution $P(y_{t'}|y_{t'-1}, x, x^{'})$. The aim of the policy gradient process is to search the policy with the maximum reward from the action space. The objective function is shown as follows:

$$\max_{\theta} L(\theta) = \max_{\theta} \mathbb{E}_{\substack{x\sim D \\ \hat{y}\sim P_\theta(\cdot|x,x^{'})}} [R(\hat{y}, x, x^{'})], \tag{11}$$

where D is the training set, θ is the parameter to be learned by the encoder-decoder model, R is the reward function, and $\hat{y}$ is the repair action sequence predicted by the model. We find the model parameters that maximize the reward by the gradient ascent method. The iterative function for training is:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta L(\theta_k), \tag{12}$$

where k is the iteration number and α is the learning rate.

The action space in the policy gradient is designed as follows: In Section 2.1, several official fix suggestions for each type of vulnerability are introduced. We design the detailed repair action according to the fix suggestion description. Besides, to improve the scalability of the model and increase the diversity of fixes to deal with some potential business-related vulnerabilities, we refer to code-related modifications [18] based on mutation operators in the field of mutation testing [22] and add these operations to the action space. These mutation operators are designed not only based on well-known mutants extensively studied in the mutation testing literature [22, 52, 58, 76], but also based on Solidity language features [38, 68]. The detailed action can be divided into 15 types, which are shown in Table 2. Each rule can be represented as $p \vdash s \rightarrow s^{'}$, which means that under the premise p, the expression, variable, keyword, or function can be changed from s to $s^{'}$ by performing a replacement. The detailed descriptions of 15 types are shown as follows:

Type 1~Type 3: The first three types are the replacement of arithmetic operators, conditional operators, and logical operators, which are popular changes in mutation testing. In the corresponding rules, the e and $e^{'}$ are the values or expressions.

Type 4: The fourth type is the operand replacement. In the corresponding rule, the var and $var^{'}$ are the operand *ope_#* or the associated operation, where # is a numeric ID indicating the order in which the operand appears in the operation statement. For example, the first operand in the operation statement can be represented as *ope_1*, the third operand can be represented as *ope_3*.

Type 5: The fifth type is the address variable replacement. The rule is to replace $tx.origin$ variable to $msg.sender$ variable, which can avoid TX vulnerability.

Type 6: The sixth type is the transfer function replacement. There are three kinds of transfer functions, $transfer(amount)$, $send(amount)$, and $call.value(amount)()$, where $amount$ represents the transfer amount. The rules are to replace $call.value(amount)()$ to $transfer(amount)$ or $send(amount)$ function and replace $send(amount)$ to $transfer(amount)$ function.

Type 7: The seventh type is ether unit replacement. Ether is the official currency used by Ethereum. Whether constructing transactions for ether transfers or invoking smart contracts for token issuance, ether is consumed. Vulnerabilities about the wrong use of ether units can easily lead to wrong transfer amounts. There are four kinds of ether units, wei, $finney$, $szabo$, and $ether$. The conversion relationship among them is $1ether = 10^3 finney = 10^6 szabo = 10^8 wei$. The rule is the replacement between them.

Type 8: The eighth type is time unit replacement. Time is one of the characteristics of smart contracts. Many smart contracts contain time constraints as conditions for contract triggering. The time units supported in Solidity are $seconds$, $minutes$, $hours$, $days$, and $weeks$. The rule is the replacement between them.

Type 9: The ninth type is the data location keyword replacement. Every reference type in Solidity has an additional data location keyword. In the internal function, there are two kinds of keywords, storage and memory. Local storage variables in smart contracts that do not have the data location keyword may be stored in unexpected storage locations, resulting in intentional or unintentional vulnerabilities. In the rule, the key and $key^{'}$ represent the keyword.

Type 10: The tenth type is variable type keyword replacement. There are four rules in this type, two for integer type and the other two for fixed-length byte array type. As for the integer, if the integer length type is between 8 and 64, then the integer type keyword is replaced with $uint64$. If the integer length type is between 64 and 256, then the integer type keyword is replaced with $uint256$. As for the fixed-length byte array, if the length type is between 1 and 7, then the variable type keyword will be replaced with $bytes8$. If the length type is between 8 and 32, then the variable type keyword will be replaced with $bytes32$.

Table 2. Action Space

Action Type	Rules
Arithmetic Operator Replacement	$▲, ▲^{'} \in \{+, -, *, /\}, ▲ \neq ▲^{'} \vdash e▲e' \rightarrow e▲^{'}e'$
Conditional Operator Replacement	$▲, ▲^{'} \in \{>, <, \geq, \leq, ==, !=\}, ▲ \neq ▲^{'} \vdash e▲e' \rightarrow e▲^{'}e'$
Logical Operator Replacement	$▲, ▲^{'} \in \{||, \&\&\}, ▲ \neq ▲^{'} \vdash e▲e' \rightarrow e▲^{'}e'$
Operand Replacement	$var = ope\_1, var^{'} \in \{ope\_2, ope\_3, ope\_1 + 1, 0\} \vdash var \rightarrow var^{'}$
	$var = 0, var^{'} = ope\_1 \vdash var \rightarrow var^{'}$
Address Variable Replacement	$var = tx.origin, var^{'} = msg.sender \vdash var \rightarrow var^{'}$
Transfer Function Replacement	$func = call.value(amount)(), func^{'} \in \{transfer(amount), send(amount)\} \vdash func \rightarrow func^{'}$
	$func = send(amount)(), func^{'} = transfer(amount) \vdash func \rightarrow func^{'}$
Ether Unit Replacement	$var, var^{'} \in \{wei, finney, szabo, ether\}, var \neq var^{'} \vdash var \rightarrow var^{'}$
Time Unit Replacement	$var, var^{'} \in \{seconds, minutes, hours, days, weeks\}, var \neq var^{'} \vdash var \rightarrow var^{'}$
Data Location Keyword Replacement	$key, key^{'} \in \{memory, storage\}, key \neq key^{'} \vdash key \rightarrow key^{'}$
Variable Type Keyword Replacement	$key = uint*, key^{'} = uint64, * \in [8, 64) \vdash key \rightarrow key^{'}$
	$key = uint*, key^{'} = uint256, * \in [64, 256) \vdash key \rightarrow key^{'}$
	$key = bytes*, key^{'} = bytes8, * \in [1, 7) \vdash key \rightarrow key^{'}$
	$key = bytes*, key^{'} = bytes32, * \in [8, 32) \vdash key \rightarrow key^{'}$
Access Domain Keyword Replacement	$key, key^{'} \in \{private, public, internal, external\}, key \neq key^{'} \vdash key \rightarrow key^{'}$
Function State Keyword Replacement	$key \in \{\varnothing, view, pure\}, key^{'} \in \{view, pure\}, key \neq key^{'} \vdash key \rightarrow key^{'}$
Special Statement Insertion	$statement \in \{revert(), assert(e), require(e)\} \vdash \varnothing \rightarrow statement$
Add Conditional Criterion	$▲ \in \{||, \&\&\} \vdash if(e) \rightarrow if(e▲e^{'})$
	$▲ \in \{||, \&\&\} \vdash require(e) \rightarrow require(e▲e^{'})$
	$▲ \in \{||, \&\&\} \vdash assert(e) \rightarrow assert(e▲e^{'})$
Move Vulnerability Line and Next Line	exchange the order of the state variable change statement and the transfer statement

Type 11: The eleventh type is access domain keyword replacement. There are four kinds of variable or function access domain keywords in Solidity: *private*, *public*, *external*, and *internal*. The rules for access domain keyword replacement are replacement between them.

Type 12: The twelfth type is function state keyword replacement. There are two kinds of function state keywords: *pure* and *view*. Declaring a function as view means that the function does not modify the state of the smart contract, i.e., it neither modifies variables nor issues transactions. A function declared pure means that the function can neither read nor modify the state. That is, the return value of the function depends only on the parameters of the function. There are three kinds of rules, that is, add function state keyword *view*, add function state keyword *pure*, and function state keyword change.

Type 13: The thirteenth type is special statement insertion. There are three kinds of statement insertion in this type, that is, *revert* statement insertion, *assert* statement insertion, and *require* statement insertion. Among these three statements, *require* and *assert* need to be followed by an e expression, which is a bool condition expression.

Type 14: The fourteenth type is adding conditional criterion in the bool condition expression. Three kinds of statements can execute this type of action, that is, if statement, *require* statement, and *assert* statement. When adding a new conditional criterion, a logical operator such as || or && is necessary. In the rule, the e and $e^{'}$ represent the bool condition expression.

Type 15: The fifteenth type is moving the vulnerability line and the next line. The rule is to exchange the order of the state variable change statement and the transfer statement.

In particular, for all the repair actions, if the number of vulnerabilities does not decrease or even increase after performing a certain action, then we will configure a negative reward for the agent to reject the action, which can reduce the state space.

The design of the reward function is considered from four aspects, namely, compilation performance, detection tool performance, code entropy, and code similarity, which is shown in Table 3. Compilation performance means judging whether the smart contract is compiled successfully. It is a prerequisite for performing all subsequent evaluations. If the fixed smart contract is compiled unsuccessfully, then a minus reward will be provided and subsequent evaluations will not be

conducted. Detection tool is used to determine whether the fixed smart contract passes the detection of vulnerability detection tools. Specifically, we utilize vulnerability detection tools to collect vulnerable smart contracts. Therefore, a positive reward is given if the number of previously detected vulnerabilities in a contract is reduced after applying the repair action. If the running time of the detection tool exceeds the time limit (we set it to 60 seconds) or the number of detected vulnerabilities does not decrease, then a negative reward is given.

Code entropy refers to an indicator of statistical code distribution, which is proposed by References [56, 61]. They utilized a statistical language model to capture the token distribution (i.e., naturalness) between vulnerable code and non-vulnerable code in traditional programming languages such as Java, C, and C++. According to the result of the statistical language model, they found that vulnerable code tended to be more entropic (i.e., unnatural). This finding provides an important indicator for finding vulnerabilities. Therefore, we adopt this approach in Solidity. Specifically, we utilize n-grams model. N-grams model is based on the Markov assumption [28], which assumes that the occurrence of the current token is only related to the previous several tokens. Therefore, we can use n-grams to predict the probability of token t_i based on the first $n-1$ tokens of t_i, which is shown as follows:

$$P(t_i|h) = P(t_i|t_{i-n+1}, \ldots, t_{i-1}), \tag{13}$$

where h is the prefix tokens. $P(t_i|h)$ can be calculated from the training corpus as the ratio of the amount that the token t_i follows that prefix token sequence $t_{i-n+1}, \ldots, t_{i-1}$:

$$P(t_i|h) = \frac{count(t_{i-n+1}, \ldots, t_{i-1}, t_i)}{count(t_{i-n+1}, \ldots, t_{i-1})}. \tag{14}$$

We also combine a cache language model ($gram), which adds a cache-list of n-grams extracted from the local context to obtain the local regularities. The local context is extracted from each test file. The combined model is illustrated as follows:

$$P(t_i|h, cache) = \lambda P(t_i|h) + (1-\lambda)P_{cache}(t_i|h), \tag{15}$$

where $P(t_i|h, cache)$ is the probability of t_i after introducing the $cache$, $P_{cache}(t_i|h)$ is t_iś probability in the n-grams model of the $cache$. λ is the interpolation weight. After getting the probability of token t_i, we can obtain the unnaturalness of token t_i measured by cross-entropy (i.e., improbability):

$$Entropy(t_i) = -P(t_i|h, cache)log_2 P(t_i|h, cache). \tag{16}$$

A higher entropy value means a lower probability and less naturalness of token t_i. As for the entropy of a code line, we compute the mean entropy of all tokens in the code line. We utilize function code without vulnerabilities in Solidity as a training corpus to train a 3-grams model and find that the entropy of vulnerable code snippets and non-vulnerable code snippets have a significant difference. The entropy of vulnerable code snippets is 4.38, while the entropy of non-vulnerable code snippets is 3.27. The detailed result is shown in Section 4.4.1. Therefore, code entropy can provide a rewarding metric for us to confirm whether the smart contract is fixed or not. If the code entropy of the smart contract after performing the repair actions becomes closer to the code entropy of the non-vulnerable smart contract, then we will give a positive reward, otherwise, we will give a negative reward.

Code similarity means the distance between vector representation of two smart contracts, which is proposed by Gao et al. [15] to detect vulnerabilities. Specifically, they parsed the smart contract code into token sequences and utilized a word embedding model called $fastText$ to encode sequences into vectors. They utilized source codes of 20,000 smart contracts to train the model and

Table 3. Objective Considered by the Reward Function

Objective	Description
Compilation Performance	Whether the fixed smart contract be compiled successfully
Detection Tool Performance	Whether the fixed smart contract passes the detection of vulnerability detection tools
Code Entropy	Whether the code distribution of fixed smart contract becomes more "natural"
Code Similarity	Whether the code vector representation of the fixed smart contract is further away from the vector representation of the vulnerable smart contract

encode 32 smart contracts with different kinds of vulnerabilities into vectors. They encoded the target smart contracts into the vector and compared them with the vulnerable smart contract vector by calculating the similarity to detect vulnerabilities. The result shows that the greater the similarity between the target smart contract vector and the specific vulnerability smart contract vector, the greater the possibility that the target contract contains the vulnerability. Therefore, we adopt this approach as a reward indicator. First, we collect source codes of 160,000 smart contracts from Etherscan and parse them into sequences to train the embedding model $fastText$. After that, we use 100 vulnerable smart contracts and extract the vulnerable line, the previous of the vulnerable line, and the next line of the vulnerable line to encode them to vectors to obtain the vulnerable vector representation. We also encode the three-line source codes of the target smart contract before fixing. Then, when we execute repair actions on the target smart contract and get the fixed smart contract, we encode the fixed three-line source code into the vector. Compared with the similarity between the target vector before fixing and the vulnerable vector, if the similarity between the fixed vector and the vulnerable vector becomes smaller, then a positive reward will be provided, otherwise, we will give a negative reward.

4 EXPERIMENTS AND EVALUATION

4.1 Data Preparation

Since there is a lack of public datasets for smart contract repair, we build a dataset of vulnerable smart contracts with five types of vulnerabilities. As mentioned in Section 3.1, we download 166,983 Solidity files from Etherscan, and the statistical characteristics are shown in Table 4. Each Solidity file has 3.96 contracts, on average (ranges from 0–75). After that, we utilize *Mythril*, *Securify*, *Oyente*, and *Slither* to detect ED, RE, TOD, IO, and TX vulnerabilities, respectively, and filter out false positive smart contracts (including benign RE) by manual checking. To verify how well the volunteers agree on the checking, we adopt Cohen's Kappa coefficient, which is a statistical measure of agreement between two evaluators [63]. The result is 0.91, which indicates good quality in manual checking. We also utilize an injection tool to increase the amount of ED and TOD vulnerabilities for training. The distribution of different types of vulnerable smart contracts is shown in Table 5. There are 196 ED vulnerable smart contracts (49 of them generated by vulnerability injection), 60 RE vulnerable smart contracts, 159 TOD vulnerable smart contracts, 219 IO vulnerable smart contracts, and 219 TX vulnerable smart contracts (130 of them generated by vulnerability injection). All vulnerable smart contracts are marked with the fault location (i.e., vulnerable line) and are public on GitHub.[8] We split 80% of each type of vulnerable smart contract into the training set and 20% into the test set. We also adopt five-fold cross-validation. The vulnerable smart contracts generated by the vulnerability injection tool will be included in the training set. Besides, we also collect 82,463 non-vulnerable functions. Specifically, since the design principle of the vulnerability detection tools is sound [48], they prefer to generate false positive results rather than false negative results. We collect all functions in which no vulnerability is reported after vulnerability

[8] https://github.com/Anonymous123xx/RLRep

Table 4. Statistical Characteristics of the Dataset

Solidity Files	No. of Contracts (Min)	No. of Contracts (Max)	Average	Contracts (Total)
166,983	0	75	3.96	661,726

Table 5. Distribution of Different Types of Vulnerable Smart Contracts

Type	Amount
ED	196 (49 of them use vulnerability injection)
RE	60
TOD	159
IO	219
TX	219 (130 of them use vulnerability injection)

detection tools detect (i.e., negative) and randomly sample 200 negative samples to execute manual validation. These samples are all true negatives. We utilize all non-vulnerable functions to train and test the statistical language model, which can validate the effectiveness of code entropy.

4.2 Experiment Configuration

4.2.1 Hardware Configuration. All the experiments were conducted on a dedicated server The server is installed with Ubuntu 21.10 Linux operating system. It also includes an Intel(R) Core(TM) i9-10980XE CPU @ 3.00 GHzx36, one GeForce RTX 3090 GPU, 256 GB memory, and 2T SSD.

4.2.2 Model Hyperparameters. We implement our approach based on PyTorch, and the corresponding hyperparameters are designed and tuned based on our experience and some research experience about reinforcement learning [9, 10]. The final values are set as follows: The *batch_size* is 64. Each encoder has 2 LSTM layers and the decoder has 1 LSTM layer. The *embedding_size* is 512 and the dictionary has 860 tokens. The *training_epoch* is 100 and the *learning_rate* is 10^{-5}. The *max_output_size* is 5, which means that the fix steps are up to 5 steps. The *beam_size* is 5, and so as for each vulnerable smart contract, our approach can provide 5 fixes. As for the design of the reward function, if the fixed smart contract cannot be compiled successfully, then we set the penalty reward as -0.02 and stop the evaluation. If the fixed smart contract is compiled successfully, then we use vulnerability detection tools to determine whether the fix is successful. We limit the detection time for each smart contract to 60 seconds, and if it times out, our approach skips this step without giving any score feedback. If it passes the detection and the fix is successful, then we reward $+0.025$, otherwise, give -0.025 penalty. After vulnerability detection, we compute the code entropy and code similarity. If the entropy of the fixed smart contract decreases, then we will give a reward $+0.014$, otherwise, a penalty -0.014 is provided. If the similarity between embedding vectors of the fixed smart contract and the vulnerable smart contract decreases, then we will give a reward $+0.014$, otherwise, a penalty -0.014 is provided. In particular, because reinforcement learning tends to be slow for the tasks with large state spaces as the time required for gathering information by state exploration increases, to reduce the agent's state search time, we label 20% of the training set with the correct sequence of repair actions as expert demonstration [2] for pre-training.

4.2.3 Test Case Verification. To confirm whether the semantics and functionality of the fix smart contract have changed and its possible impact, we also need some test cases to execute regression test to verify the reliability of the fixed smart contract. Because there is a lack of test cases

used for smart contracts on the blockchain [44, 75], we utilize the transactions invoking the target contract to generate regression test cases. These transactions are public on Etherscan. Specifically, for a smart contract executing the repair actions, we extract several transactions invoking their corresponding original smart contracts. Each transaction is denoted by $t_i (i = 1, 2, \cdots)$. We replay the transaction t_i and capture the inputs and the changes to the transaction-related blockchain state (i.e., the account balance on the blockchain) during the execution. The inputs and changes can be considered as the input and expected behaviors of the generated regression test cases. A regression test case generated from a transaction t_i include four elements:

(1) Transaction-related blockchain state before executing the transaction t_i.
(2) Transaction-related blockchain state after executing the transaction t_i.
(3) The function to be called and the corresponding parameter value.
(4) The return value of the called function.

If the fixed smart contract behaves as expected after executing the regression test case, then it can be regarded as a repair recommendation without changing function.

4.2.4 Manual Verification. Due to the accuracy of the model, we invite two new volunteer students (different from students checking the smart contract dataset) who have two years of Solidity development experience to verify that the contract that performed the repair action was correctly fixed. Specifically, the two students determine whether the repaired contracts were repaired as suggested based on the contract repair recommendations provided by the SWC registry and previous work. The detailed contract repair recommendations are provided in Section 2.1. If both developers agree the contract that performs the repair action is fixed according to the SWC registry's and previous work's recommendations, and both agree that the original contract semantics are not changed, then the fix is correct. If both developers agree that the fix is faulty, then the fix is not the correct patch. If there is a discrepancy between the two evaluations, then the final result is determined through discussion.

4.3 Evaluation

We conduct experiments to evaluate the validity and performance of *RLRep*. We focus on four research questions:

RQ1: What is the performance of RLRep in the repair recommendation? We evaluate *RLRep* in providing repair recommendations of typical vulnerabilities. The result is shown in Table 6. There are 171 vulnerable smart contracts with typical vulnerabilities in the test set. Since we set the *beam_size* as 5, every vulnerability has 5 repair recommendations, and the number of total repair recommendations is 855. We regard the fixed smart contract that can be compiled as the compilable patch. If there is a repair recommendation that can make the target smart contract pass the detection of detection tools, test case verification, and manual verification in five repair recommendations, then we consider the target smart contract as fixable, i.e., a correct patch will be recorded. Therefore, a correct patch corresponds to a vulnerable smart contract that can be repaired. The detailed process of test case verification and manual verification is shown in Section 4.2.3 and Section 4.2.4. The CohenâĂŹs Kappa coefficient result of manual verification in this experiment is 0.87, which indicates the good quality of manual verification.

It can be found that *RLRep* can provide 266 compilable repair recommendations in total and 94 of them are the correct patch. In particular, these correct patches correspond to different vulnerabilities. And as for test case verification, we replay the transactions on the first 9 million blocks on Ethereum and record the blockchain state before and after the execution of the transactions invoking our target contracts (i.e., smart contracts in the test set). We find that each of our correct repair contracts is invoked by at least one transaction in these 9 million blocks. The number of

Table 6. Performance of *RLRep* in Repair Recommendation of Typical Vulnerabilities

Vulnerability Type/ No. of Test Set	*RLRep*			Similarity-reverted *RLRep*		
	Compilable Patch	Correct Patch	Correct Patch (%)	Compilable Patch	Correct Patch	Correct Patch (%)
ED/36	35	18	50.00%	27	7	19.44%
RE/10	13	8	80.00%	11	8	80.00%
TOD/37	11	0	0.00%	29	0	0.00%
IO/45	87	29	64.44%	154	37	82.22%
TX/43	120	39	90.70%	128	37	86.05%
Total/171	266	94	54.97%	349	89	52.05%
Vulnerability Type/ No. of Test Set	Transformer Structure *RLRep*			The Approach proposed in [75]		
	Compilable Patch	Correct Patch	Correct Patch (%)	Compilable Patch	Correct Patch	Correct Patch (%)
ED/36	46	18	50.00%	107	8	22.22%
RE/10	21	8	80.00%	22	0	0.00%
TOD/37	13	0	0.00%	62	0	0.00%
IO/45	61	0	0.00%	189	0	0.00%
TX/43	79	37	86.05%	199	0	0.00%
Total/171	220	63	36.84%	579	8	4.68%

correct repair contracts is 83, and these contracts include all 94 correct patches (a contract may contain multiple vulnerabilities). In the replay results, there is no change in output and blockchain state for all test cases when invoking the smart contract before and after the fix. This indicates that the fix recommendations provided by *RLRep* do not change the semantics and functionality of existing transactions (i.e., the functionality that the contract user wants to implement with that contract) in terms of preventing potential security.

As for each vulnerability type, there are 35, 13, 11, 87, and 120 compilable recommendations for each vulnerability type. Thirty-nine TX vulnerabilities have the correct repair recommendations, which is the vulnerability type with the most correct repair recommendations. In contrast, there is no compilable recommendation or correct repair recommendation generated for the TOD vulnerability. The reason may be that the business and semantics logic of TOD vulnerabilities are more complex than other vulnerabilities, and the number of current data for TOD vulnerabilities is relatively limited, making it difficult to learn complex repair actions. Besides, the percentage of correct repair recommendations for each vulnerability type is 50.00%, 80.00%, 0%, 64.44%, and 90.70%, respectively.

Some references [34, 67] proposed that a patch that is more similar to the vulnerable program is more likely to be correct, since the more similar semantics. To analyze the relationship between code similarity variation and smart contract repair recommendation results, we modify the revert code similarity reward of the reward function in our approach, i.e., we reward the repair actions that make the fixed contract similar to the vulnerable contract. We call this approach similarity-reverted *RLRep*. The results show that the numbers of correct repair recommendations in RE, TOD, IO, and TX are similar, but the number of ED correct repair recommendations is less than *RLRep*. This suggests that the fix for the ED vulnerability may involve a larger code change and rewarding a similarity decrease has a better performance.

Transformer [64] is a state-of-the-art architecture that was proposed in 2017 to replace RNN and LSTM networks in different NLP tasks. We also adopt this structure in our approach to replace LSTM network. We call this approach Transformer structure *RLRep*. The results show that

the numbers of correct repair recommendations in ED, RE, TOD, and TX are similar, but this structure cannot provide correct repair recommendations for IO vulnerabilities as *RLRep* does. The results indicate that Transformer may not be suitable for smart contract repair recommendation. It may be because Transformer has a larger amount of model parameters and the hyperparameters are difficult to adjust. Besides, Transformer lacks modeling of the time dimension, but the code sequences are sequential.

In addition, we also compare *RLRep* with a search-and-genetic-algorithm-based smart contract repair approach proposed in Reference [75]. This approach had been the first work on automatically repairing smart contracts, as the authors declared [75]. Besides, it applied smart contract source code as model input, which is similar to our approach (source code + AST). Since the public code of this article on GitHub cannot be executed and the authors of the paper could not be contacted, we reproduce their method according to the approach description in the paper and adopt the fix result on our test set as a baseline. The reproduced code is also public on GitHub.[9] For each sample, we set the generation size of the genetic algorithm to 15. And we select the five individuals with the best performance in fitness function in the last generation population of the genetic algorithm as the repair recommendation. To ensure the fairness of the optimization metric, we set the objectives of the fitness function to be the same as that in the reward function of *RLRep*. With fixes on the entire test set, the algorithm's running time is 43 days, which is much longer than *RLRep* (56 hours). The results show that this search-and-genetic-algorithm-based repair approach can generate more compilable patches, because the uncompilable patches are filtered out during mutation. However, this approach can only provide 8 correct patches for ED vulnerable smart contracts, which is 125% less than *RLRep*. It cannot provide fix recommendations for some other complex vulnerabilities, such as RE, TOD, IO, and TX. But *RLRep* can fix some parts of RE, IO, and TX vulnerabilities. In particular, as for the 17 vulnerable smart contracts proposed in Reference [75], because this paper only provided the vulnerable smart contracts but did not provide other information such as vulnerability location and the vulnerability number is different from our detection results, we can only include true-positive samples of the 17 smart contracts in our dataset based on our vulnerability detection results. In other words, *RLRep* has a better performance in repair recommendation.

RQ2: What are the effects of different modules in reward function on the performance of code repair recommendation? As mentioned in Section 3.3, we adopt compilation performance, detection tool performance, code entropy, and code similarity to construct the reward function in the policy gradient algorithm. To analyze the influence of these four objects on repair recommendation, we conduct an ablation study. Since we set the compilation result as the prerequisite for performing other modules in the reward function, we evaluate the performance of providing repair recommendations by comparing *RLRep* with three policy gradient models with different reward functions: (1) the model with the reward function considering compilation and detection tool performance; (2) the model with the reward function considering compilation and code entropy; (3) the model with the reward function considering compilation and code similarity; (4) the model with the reward function considering compilation, code similarity, and detection tool performance. The result is shown in Table 7. It can be found that the different modules in the reward function all help the repair recommendation of the contract to different degrees, but if the different modules are combined together, then their recommendation effect has different degrees of improvement compared with the effect of individual modules (up to 29 correct patches are added). This result demonstrates the effectiveness of the reward function design in *RLRep*.

RQ3: What are the effects of context on the performance of code repair recommendation? As mentioned in Section 3.1, we extract the previous line and the next line of the vulnerable line to build

[9] https://github.com/Anonymous123xx/RLRep/blob/main/genetic.py

Table 7. Influence of Four Objects Considered in the Reward Function on Repair Recommendation

Reward Function Object	Compilable Patch	Correct Patch
Compilation & Detection Tool	261	65
Compilation & Code Entropy	422	76
Compilation & Code Similarity	345	79
Compilation & Code Similarity & Detection Tool	341	88
RLRep	266	94

Table 8. Influence of Context on Repair Recommendation

Context Scope	Compilable Patch	Correct Patch
The Entire Function	251	81
The Previous Two lines & the Next Two Lines	337	84
The Previous Line Only	282	86
The Next Line Only	344	73
No Context	257	72
RLRep	266	94

the three-line vulnerable code snippet as the model input. We analyze how this context information affects the performance of code repair recommendation. The results are shown in Table 8. It can be found that the result of the vulnerable line + the previous two lines & the next two Lines (84 correct patches) is worse than *RLRep* (94 correct patches). The result of the entire vulnerable function (81 correct patches) is worse than that of *RLRep*, too. The results illustrate that too much contextual information (e.g., vulnerable line + the previous two lines & the next two Lines, the entire vulnerable function) causes redundancy in the model input information, which in turn causes a decrease in the performance of repair recommendations. We proposed several detailed examples to show the influence of redundancy. In Figure 8, *RLRep* can generate a correct repair recommendation for IO vulnerability in line 10 based on three lines input. If the input is the previous two lines and the next two lines of the vulnerable line or the entire function, then the suggested repair action with the highest probability is address variable replacement (replace *tx.origin* to *msg.sender*). The generation of this incorrect action may be influenced by the *msg.sender* token in line 5. Another example is shown in Figure 9. *RLRep* can generate a correct repair recommendation for RE vulnerability in line 5. If there exists redundancy input, then the token that replaces *call.value* function is *transfer* instead of *send*. Although *transfer* function can also address RE vulnerability, it throws an exception instead of returning a *false* value once the transaction fails (gas exceeded). Therefore, it does not apply to *if* statement. The generation of this incorrect action may be influenced by the *TransferLog* token in line 9. In summary, redundant inputs have a negative impact on the generation of repair recommendations. However, if there is no context or only contains information from the previous or next line, then the performance of repair recommendation is also not good. Therefore, we extract the previous line and the next line of the vulnerable line as the context.

RQ4: Do the repair recommendation have a significant impact on the change in gas consumption? As mentioned in Section 4.2.3 and RQ1, we adopt transactions that invoke the target smart contract as regression test cases and replay them to verify whether the function and semantics of the smart contract has changed. Besides, we also analyze the difference in gas consumption when the test case invokes the pre-fixed and post-fixed smart contracts. In the first 9 million blocks, our smart contract dataset involves a total of 58,956 historical transactions. We calculate the average

```
contract UADA {
  ...
  function transferFrom(address _from, address _to, uint256 _value)
  public returns (bool success)
  {
    uint256 allowance = allowed[_from][msg.sender];
    require(balances[_from] >= _value && allowance >= _value);
    balances[_to] += _value; \\vulnerable line
+   require(_value <= balances[_to]); \\fixed by RLRep
    balances[_from] -= _value;
    if (allowance < MAX_UINT256) {
      allowed[_from][msg.sender] -= _value;
    }
    Transfer(_from, _to, _value);
    return true;
  }
  ...
}
```

Fig. 8. IO vulnerability that *RLRep* can fix rather than redundancy input.

```
contract PrivateBank {
  ...
  function CashOut(uint _am) {
    if(_am<=balances[msg.sender]) {
-     if(msg.sender.call.value (_am))() { \\vulnerable line
+     if(msg.sender.send (_am)){ \\fixed by RLRep
+     if(msg.sender.transfer (_am)){ \\generated by the redundancy input
       balances[msg.sender]-=_am;
       TransferLog.AddMessage(msg.sender,_am,"CashOut");
      }
    }
  }
  ...
}
```

Fig. 9. RE vulnerability that *RLRep* can fix rather than redundancy input.

gas usage for the transactions invoking each target smart contract, i.e., the total gas consumption of all transactions invoking the target contract divided by the number of transactions. Figure 10 depicts the change of gas consumption after adopting repair recommendations. There are 45 repaired smart contracts with reduced gas consumption. In addition, more than 84.15% (69 smart contracts) of the repaired smart contracts have no more than 20% increase in gas consumption. Overall, the average gas consumption per transaction is reduced by 73.32 units (i.e., 0.0027 USD[10]). This clearly demonstrates *RLRep* can provide repair economical repair recommendations, because the gas consumption increments are negligible.

Moreover, we also analyze the relationship between vulnerability type and gas consumption. The result is shown in Table 9. It can be found that repair recommendations that cause gas usage change of less than 20% are mainly IO, TX vulnerabilities. These recommendations are lightweight changes such as adding a *require* statement and replacing *tx.origin*. For a small number of fixes that cause drastic changes in gas consumption, on the one hand, this may be due to drastic code changes (e.g., RE repair recommendations). On the other hand, for a specified contract with significant gas change, the transaction gas consumption before executing the repair action is inherently small, and its increment rate will be relatively large for the same absolute increment after the repair

[10]This price is calculated by the average gas price and the Ethereum (ETH) price on February 15, 2023.

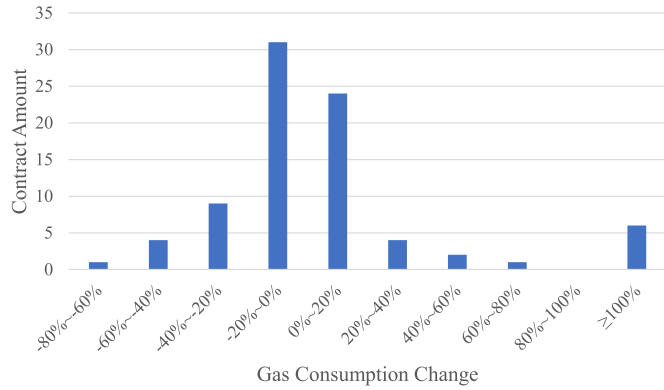

Fig. 10. Gas consumption change for repaired contracts.

Table 9. Relationship between Gas Usage and Vulnerability Type

Gas Consumption Change	Vulnerability Type				
	ED	RE	TOD	IO	TX
±20%	14	3	0	20	18
>+100%	2	1	0	0	3
<-40%	0	0	0	3	2

action is executed. In general, however, for the different vulnerability types, most of the changes in gas consumption are within acceptable limits.

4.4 Discussion

4.4.1 Code Entropy Validity. To verify the validity of code entropy as an indicator of the reward function, we collect 82,463 non-vulnerable functions and split 80% of them to train a 3-gram statistical language model combined with the cache. Then, we utilize this trained model to compute the entropy of the remaining 20% of the non-vulnerable functions and the entropy of the vulnerable functions in the vulnerable smart contract we collected (the vulnerable function in the vulnerable smart contract can be located by the vulnerability detection tool). The result is shown in Table 10. The mean entropy of different kinds of vulnerable functions, the total vulnerable functions, and the non-vulnerable functions are illustrated. It can be shown that the mean entropy of vulnerable functions is 4.38, while the mean entropy of non-vulnerable functions is 3.27. Besides, the mean entropy of vulnerable functions is less than that of non-vulnerable functions with statistical significance (p-value = 1.62×10^{-59}). Therefore, there is a significant difference between them, and code entropy can be used as one of the indicators to judge whether the smart contract has been effectively fixed.

4.4.2 The Design of Reward Value Hyperparameters. As mentioned in Section 4.2.2, we design and tune the reward and penalty value of the four objectives in the reward function based on our experience and reinforcement learning research. In this section, we present the results of several different sets of value hyperparameter designs, which are shown in Table 11. For example, as for the first line, the design is shown as follows: If the fixed smart contract cannot be compiled successfully, then we set the penalty reward as −0.02 and stop the evaluation. If the fixed smart contract is compiled successfully, then we use vulnerability detection tools to determine whether the fix is successful. If it passes the detection and the fix is successful, then we reward +0.02,

Table 10. Entropy Difference between Vulnerable Function and Non-vulnerable Function in Solidity

Type	Entropy
ED	4.70
RE	6.39
TOD	4.64
IO	3.19
TX	4.36
Total Vulnerable	4.38 (**p<0.01)
Total Non-vulnerable	3.27

Table 11. Results of Different Reward Value Designs

Reward & Penalty Value				Compilable Patch	Correct Patch
Unable to compile	Detection Tool Performance	Code Entropy	Code Similarity		
−0.02	±0.02	±0.01	±0.01	240	64
−0.04	±0.04	±0.02	±0.02	233	60
−0.06	±0.06	±0.03	±0.03	218	70

otherwise, give −0.02 penalty. After vulnerability detection, we compute the code entropy and code similarity. If the entropy of the fixed smart contract decreases, then we will give a reward +0.01, otherwise, a penalty −0.01 is provided. If the similarity between embedding vectors of the fixed smart contract and the vulnerable smart contract decreases, then we will give a reward +0.01, otherwise, a penalty −0.01 is provided. The designs of the second line and the third line are similar. It can be found that all the results from these designs are not better than the final design.

4.4.3 Illustrative Example. In this section, we present several concrete cases of fixes that illustrate the performance of *RLRep*. As for each case, we select the correct recommendation in the five recommendations and illustrate it. As shown in Figure 11, case 1 shows an example of the IO vulnerability repair recommendation. It can be found that a *require* statement is inserted in line 17. This statement can be generated by executing the actions *Special Statement Insertion* and *Operand Replacement*. Without this statement, an IO vulnerability may be triggered in line 16. Case 2 in Figure 12 shows an example of RE vulnerability repair recommendation. In line 6 of the code snippet, the utilization of *call.value* function may cause RE vulnerability. The approach proposes the recommendation by executing action *Transfer Function Replacement* to replace *call.value* function to *send* function. The examples show that *RLRep* can suggest correct fixes for some typical vulnerabilities.

5 RELATED WORK

5.1 Automatic Program Repair

Many techniques have been adopted in automatic program repair such as heuristic search, defined repair patterns, and machine learning [19]. As for heuristic search, the genetic algorithm is the most popular algorithm. Goues et al. proposed a genetic algorithm-based automatic program repair approach, GenProg [35]. This approach converted the source code developed by #C language into an **AST (abstract syntax tree)**. Then, it iteratively used crossover operators and mutation operators to randomly delete, add, or replace nodes in the existing AST to generate a new AST and converted it into the corresponding program patch, and finally performed patch verification. After that, they proposed to utilize this approach to repair large applications [36] and found that

```
contract Burner {
  address public dragon;
  uint256 public DragonsBurned;
  modifier onlyDragon() {
    if (msg.sender != dragon) {
      throw;
    }
    _;
  }
  function Burner () {
    dragon = 0x814F67fA286f7572B041D041b1D99b432c9155Ee;
  }
  function dragonHandler(uint256 _amount) onlyDragon {
    Dragon drag = Dragon(dragon);
    drag.burnDragons(_amount);
    DragonsBurned += _amount;
+   require(_amount <= DragonsBurned);
  }
}
```

Fig. 11. Case 1: IO vulnerability repair recommendation.

```
contract I_BANK {
  ...
  function Collect(uint _am) public payable {
    var acc = Acc[msg.sender];
    if(acc.balance>=MinSum && acc.balance>=_am && now>acc.unlockTime) {
-     if(msg.sender.call.value(_am)()) {
+     if(msg.sender.send (_am)) {
        acc.balance -= _am;
        LogFile.AddMessage(msg.sender, _am, "Collect");
      }
    }
  }
  function() public payable {
    Put(0);
  }
  ...
}
```

Fig. 12. Case 2: RE vulnerability repair recommendation.

GenProg could fix 55 of 105 defects in large applications. There was much follow-up research work after GenProg such as RSRepair [55], Nopol [70], and so on. Defined repair patterns-based automatic repair approaches refer to template-based program repair. Liu et al. [42] first surveyed the existing literature and collected, summarized, and labeled various types of repair templates that are frequently used. Based on this survey, they proposed *TBar*, which systematically attempted to fix program errors directly using these repair templates. The fix reflected a performance unprecedented in the prior literature. The machine learning-based or data-driven automatic program repair is popular recently. Tufano et al. [62] proposed that the encoder-decoder model could be used to learn meaningful code changes including the vulnerability fix. Jiang et al. [26] proposed a Java program repair approach that combined GPT and encoder-decoder models. This approach generated more compilable patches and correct patches than baselines on both Defect4J and QuixBugs vulnerability libraries. Zhu et al. [81] proposed to use the transformer-based model to encode some modified rules (a series of generators), the code with the vulnerability, and the syntax tree corresponding to the code to generate a patch for the fix. The approach was evaluated on Defect4J and provided good repair results even without perfect defect localization. He et al. [74] proposed

an NMT-based program repair approach based on supervised learning. It consisted of three parts: syntactic training, semantic learning, and inference. In particular, each semantic training sample is labeled with a reward considering program compilation and test execution information. The model parameters are updated by calculating reward loss. Some approaches working without labeled data were also presented recently. For example, Yasunaga et al. [72] proposed to train a breaker and a fixer to generate fix-and-broken code pairs from unlabeled real good codes and bad codes. He et al. [73] proposed a program repair approach based on self-supervised training. They added a perturbation to build a vulnerability dataset from a correct version of the project in GitHub and encoded both source codes and test execution diagnostics to generate patches. The approaches mentioned above aimed to fix vulnerabilities in traditional programming languages (e.g., Java, #C, and Python). Some approaches adopted unlabelled data. In this article, we propose a program repair recommender for the smart contract vulnerability fix. We mainly utilize the vulnerable code collected from real smart contracts deployed in Ethereum to train the fix recommendation models directly rather than adding perturbation or using models to generate buggy-and-fix code pairs. We adopt a reward function to evaluate the fixed codes that not only considers the syntax knowledge but also the business logic vulnerability in the policy gradient training.

5.2 Security Analysis of Smart Contract

Because smart contracts involve many finance-related businesses such as asset transfers and cannot be tampered with once they are deployed on the blockchain, it is crucial to ensure the security of smart contracts. There are many studies related to smart contract security analysis. One kind of research investigates and analyzes the types of vulnerabilities in smart contracts. For example, Praitheeshan et al. [54] investigated 16 security vulnerabilities in smart contract programs and predicted that many attacks against smart contracts had not been exploited. This survey can help smart contract developers to have a more comprehensive understanding of security issues. Kushwaha et al. [33] presented a systematic review of Ethereum smart contract security vulnerabilities and the corresponding detection tools and preventive mechanisms. This survey could provide research direction in the future for researchers. In our article, we focus on five kinds of vulnerabilities and propose an approach to provide repair recommendations for these five kinds of vulnerabilities.

Another kind of security analysis research of smart contracts focuses on proposing security analysis tools or vulnerability detection tools. These tools are developed based on different approaches such as symbolic execution, abstract interpretation, formal verification, machine learning, and so on. For instance, Luu et al. [44] proposed to analyze EVM bytecode symbolically to detect timestamp dependence, transaction-ordering dependence, exception disorder, and reentrancy vulnerabilities in smart contracts. Tsankov et al. [60] proposed to define rules of security properties and determine whether a smart contract is vulnerable by detecting whether it violates the defined security property rules. Bhargavan et al. [5] proposed to translate Solidity source code to formal language to detect the runtime safety and functional correctness of smart contracts. Gao et al. [15] proposed to use word embedding to encode code tokens of smart contracts and utilize similarity detection to detect smart contract vulnerabilities. In our article, we adopt the result of several detection tools as the metrics of the reward function.

5.3 Repair Recommendation of Smart Contract

Nowadays, there are two kinds of repair approaches for the smart contract: on-chain repair and off-chain repair. On-chain repair means making security updates to smart contracts already deployed to the blockchain. For example, Rodler et al. [57] proposed a bytecode rewriting engine for Ethereum blockchain called EVMPATCH, which could rewrite the vulnerable smart contract to an upgradeable smart contract. This approach can repair integer overflow and access control

vulnerabilities. Jin et al. [27] proposed an on-chain smart contract repair framework to repair integer overflow, reentrancy, and unchecked low-level checks vulnerabilities. They utilized coarse-grained inverse slicing and data flow analysis techniques to extract key information of the smart contracts and generated the repaired smart contract based on program synthesis techniques. They also proposed a novel contract operation mechanism to realize the binding of vulnerable contracts and fixed contracts to defend vulnerability attacks.

Off-chain repair means providing more secure smart contract repair solutions before contracts are deployed on the blockchain (usually during the contract audit phase). This method allows for direct code modification on the original smart contract. For instance, Zhang et al. [77] designed a bytecode repair system called SMARTSHIELD to repair three typical security-related vulnerabilities (i.e., state changes after external calls, missing checks for out-of-bound arithmetic operations, and missing checks for failing external calls). They designed rewriting rules for these three vulnerabilities at the bytecode level through semantic extraction to attain secure EVM bytecode. Torres et al. [59] also proposed a scalable approach toward automatic smart contract repair at the bytecode level. The proposed approach adopted template-based and semantic-based patching by inferring context information from bytecode and it could patch seven different types of vulnerabilities. Nguyen et al. [48] proposed a pattern-based smart contract fixing method for four common kinds of vulnerabilities. Their approach used runtime information and was proved to be sound. Yu et al. [75] proposed an automated smart contract repair approach based on a search-and-genetic-based algorithm. Specifically, it searches among mutations of the vulnerable contract source code. Besides, it was a gas-aware approach. In our article, we propose an off-chain repair recommendation approach based on reinforcement learning, which can provide repair recommendations for more types of vulnerabilities.

6 LIMITATION AND THREATS TO VALIDITY

Limitation. The repair actions we design in *RLRep* can not provide repair recommendations for all types of vulnerabilities. With the development of blockchain-related business (e.g., digital asset trading, DeFi), new types of smart contract vulnerabilities have emerged. These new types of vulnerabilities often have more complex patterns and logic and are difficult to repair with specific rules. Therefore, our approach is currently unable to provide repair recommendations for these vulnerabilities, that is, the limitation of scalability in new vulnerability types. Besides, *RLRep* mainly focuses on function-local vulnerabilities now; some cross-function or cross-contract vulnerabilities have not been considered yet. Nonetheless, *RLRep* can provide repair recommendations for new smart contract developers to avoid typical smart contract vulnerabilities. In the future, we will try to summarize new vulnerability fixing rules and add to the action space when possible to improve the scalability. We will also consider some other code features such as call graphs as the model input to address the issue of the cross-function or cross-contract vulnerability fix.

Threats to external validity. The threats to external validity refer to the scalability of *RLRep*. *RLRep* utilizes code snippets written in Solidity language, but there are some other smart contract languages such as Vyper, Yul, and so on. The code features and vulnerability patterns are different. These kinds of features should be carefully processed when inputting the AST or source code. The action space is also necessary to be redesigned. Therefore, more investigation by analyzing codes written by other smart contract languages should be executed.

Threats to internal validity. The threats to internal validity refer to the scale of the dataset and the design of the reward function. In this article, we utilize vulnerability detection tools and the vulnerability injection tool to collect 853 vulnerable smart contracts. The size is limited, which may cause a biased of the model result. Besides, although we use the vulnerability injection tool [17] to increase the number of certain vulnerability types, the number of samples for some vulnerabilities

(e.g., IO and TOD) types is still limited due to the limitations of the tool. As the size of the number of smart contracts increases, it is necessary to increase the size of the vulnerable smart contract dataset for each type. In addition, we utilize the result of vulnerability detection tools as one of the factors considered in the reward function. The accuracy of the vulnerability detection tools may influence the performance of the repair recommendation. To address this issue, we provide five repair recommendations for each vulnerability and adopt manual verification to confirm the correct patch.

7 CONCLUSION

Smart contract fixes play a critical role in developing smart contracts and maintaining their security. In this article, we propose an approach called *RLRep* to provide smart contract repair recommendations automatically based on reinforcement learning. We use smart contracts with vulnerabilities as state inputs and a series of fix rules as action outputs to solve the problem of unlabeled data of vulnerable contracts. In addition, we use beam search to generate multiple fix recommendations for each contract. We train and test our model on five vulnerability types with 853 vulnerability smart contracts. We demonstrate that *RLRep* can generate correct repair recommendations with better results than genetic algorithm-based repair approaches.

REFERENCES

[1] Oxford Analytica. 2021. Poly network attack underlines growing DeFi risks. *Emerald Expert Briefings* oxan-es (2021).

[2] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. 2009. A survey of robot learning from demonstration. *Robot. Auton. Syst.* 57, 5 (2009), 469–483. DOI: https://doi.org/10.1016/j.robot.2008.10.024

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. DOI: https://doi.org/10.1145/3360585

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[5] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security (PLAS@CCS'16)*, Toby C. Murray and Deian Stefan (Eds.). ACM, 91–96. DOI: https://doi.org/10.1145/2993600.2993611

[6] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. 2013. Reversible Debugging Software "Quantify the Time and Cost Saved Using Reversible Debuggers." (2013).

[7] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. Defining smart contract defects on ethereum. *IEEE Trans. Softw. Eng.* 48, 1 (2022), 327–345. DOI: https://doi.org/10.1109/TSE.2020.2989002

[8] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. 2018. Smart contracts vulnerabilities: A call for blockchain software engineering? In *Proceedings of the International Workshop on Blockchain Oriented Software Engineering (IWBOSE'18)*. 19–25. DOI: https://doi.org/10.1109/IWBOSE.2018.8327567

[9] Rati Devidze, Goran Radanovic, Parameswaran Kamalaruban, and Adish Singla. 2021. Explicable reward design for reinforcement learning agents. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 20118–20131. Retrieved from https://proceedings.neurips.cc/paper/2021/file/a7f0d2b95c60161b3f3c82f764b1d1c9-Paper.pdf

[10] Daniel Dewey. 2014. Reinforcement learning and the reward engineering principle. In *Proceedings of the AAAI Spring Symposium Series*.

[11] ConsenSys Diligence. 2018. Ethereum smart contract security best practices. Retrieved from https://consensys.github.io/smart-contract-best-practices/

[12] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *Proceedings of the IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB'19)*. 8–15. DOI: https://doi.org/10.1109/WETSEB.2019.00008

[13] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. The eye of Horus: Spotting and analyzing attacks on ethereum smart contracts. In *Financial Cryptography and Data Security*, Nikita Borisov and Claudia Diaz (Eds.). Springer Berlin, 33–52.

[14] Dr Catherine Flick. 2022. A critical professional ethical analysis of Non-Fungible Tokens (NFTs). *J. Respons. Technol.* 12 (2022), 100054. DOI: https://doi.org/10.1016/j.jrt.2022.100054

[15] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John Grundy. 2021. Checking smart contracts with structural code embedding. *IEEE Trans. Softw. Eng.* 47, 12 (2021), 2874–2891. DOI: https://doi.org/10.1109/TSE.2020.2971482

[16] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Trans. Softw. Eng.* 45, 1 (2019), 34–67. DOI: https://doi.org/10.1109/TSE.2017.2755013

[17] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. Association for Computing Machinery, New York, NY, 415–427. DOI: https://doi.org/10.1145/3395363.3397385

[18] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. *Practical Program Repair via Bytecode Mutation*. Association for Computing Machinery, New York, NY, 19–30. DOI: https://doi.org/10.1145/3293882.3330559

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65. DOI: https://doi.org/10.1145/3318162

[20] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. Association for Computing Machinery, New York, NY, 531–548. DOI: https://doi.org/10.1145/3319535.3363230

[21] Gang Huang, Chaoran Luo, Kaidong Wu, Yun Ma, Ying Zhang, and Xuanzhe Liu. 2019. Software-defined infrastructure for decentralized data lifecycle governance: Principled design and open challenges. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS'19)*. IEEE, 1674–1683.

[22] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.* 37, 5 (2011), 649–678. DOI: https://doi.org/10.1109/TSE.2010.62

[23] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. 259–269. DOI: https://doi.org/10.1145/3238147.3238177

[24] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. 255–266. DOI: https://doi.org/10.1109/ASE.2019.00033

[25] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. Association for Computing Machinery, New York, NY, 298–309. DOI: https://doi.org/10.1145/3213846.3213871

[26] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. 1161–1173. DOI: https://doi.org/10.1109/ICSE43902.2021.00107

[27] Hai Jin, Zeli Wang, Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou. 2021. Aroc: An automatic repair framework for on-chain smart contracts. *IEEE Trans. Softw. Eng.* (2021), 1–1. DOI: https://doi.org/10.1109/TSE.2021.3123170

[28] J. D. Kalbfleisch and J. F. Lawless. 1985. The analysis of panel data under a Markov assumption. *J. Amer. Statist. Assoc.* 80, 392 (1985), 863–871. DOI: https://doi.org/10.1080/01621459.1985.10478195

[29] JingHuey Khor, Mansur Aliyu Masama, Michail Sidorov, WeiChung Leong, and JiaJun Lim. 2020. An improved gas efficient library for securing iot smart contracts against arithmetic vulnerabilities. In *Proceedings of the 9th International Conference on Software and Computer Applications (ICSCA'20)*. Association for Computing Machinery, New York, NY, 326–330. DOI: https://doi.org/10.1145/3384544.3384577

[30] Ki Byung Kim and Jonghyup Lee. 2020. Automated generation of test cases for smart contract security analyzers. *IEEE Access* 8 (2020), 209377–209392. DOI: https://doi.org/10.1109/ACCESS.2020.3039990

[31] Philipp Koehn. 2004. Pharaoh: A beam search decoder for phrase-based statistical machine translation models. In *Machine Translation: From Real Users to Research*, Robert E. Frederking and Kathryn B. Taylor (Eds.). Springer Berlin, 115–124.

[32] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, Baltimore, MD, 1317–1333. Retrieved from https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[33] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access* 10 (2022), 6605–6621. DOI: https://doi.org/10.1109/ACCESS.2021.3140091

[34] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software*

Engineering (ESEC/FSE'17). Association for Computing Machinery, New York, NY, 593–604. DOI: https://doi.org/10.1145/3106237.3106309

[35] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 3–13. DOI: https://doi.org/10.1109/ICSE.2012.6227211

[36] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 54–72. DOI: https://doi.org/10.1109/TSE.2011.104

[37] Yuzheng Li, Chuan Chen, Nan Liu, Huawei Huang, Zibin Zheng, and Qiang Yan. 2021. A blockchain-based decentralized federated learning framework with committee consensus. *IEEE Netw.* 35, 1 (2021), 234–241. DOI: https://doi.org/10.1109/MNET.011.2000263

[38] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. 2019. MuSC: A tool for mutation testing of ethereum smart contract. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. 1198–1201. DOI: https://doi.org/10.1109/ASE.2019.00136

[39] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-aware code change embedding for better patch correctness assessment. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 51 (May 2022), 29 pages. DOI: https://doi.org/10.1145/3505247

[40] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding reentrancy bugs in smart contracts. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. 65–68.

[41] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, TegawendÃĪ F. BissyandÃĪ, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*. 275–286. DOI: https://doi.org/10.1109/ICSME.2018.00037

[42] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. Association for Computing Machinery, New York, NY, 31–42. DOI: https://doi.org/10.1145/3293882.3330577

[43] Xuanzhe Liu, Gang Huang, Qi Zhao, Hong Mei, and M. Brian Blake. 2014. iMashup: A mashup-based framework for service composition. *Sci. China Inf. Sci.* 57, 1 (2014), 1–20. DOI: https://doi.org/10.1007/s11432-013-4782-0

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

[45] Adrian Manning. 2018. Solidity security: Comprehensive list of known attack vectors and common anti-patterns. *Sigma Prime* 20, 10 (2018). Retrieved from https://github.com/sigp/solidity-security-blog

[46] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. 2016. Deep reinforcement learning: An overview. In *Proceedings of SAI Intelligent Systems Conference (IntelliSys'16) (Lecture Notes in Networks and Systems)*, Yaxin Bi, Supriya Kapoor, and Rahul Bhatia (Eds.), Vol. 16. Springer, 426–440. DOI: https://doi.org/10.1007/978-3-319-56991-8_32

[47] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterd.* 9 (2018), 54.

[48] Tai D. Nguyen, Long H. Pham, and Jun Sun. 2021. SGUARD: Towards fixing vulnerable smart contracts automatically. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'21)*. 1215–1229. DOI: https://doi.org/10.1109/SP40001.2021.00057

[49] 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 (ICSE'20)*. Association for Computing Machinery, New York, NY, 778–788. DOI: https://doi.org/10.1145/3377811.3380334

[50] Yuanping Nie, Yi Han, Jiuming Huang, Bo Jiao, and Aiping Li. 2017. Attention-based encoder-decoder model for answer selection in question answering. *Front. Inf. Technol. Electron. Eng.* 18, 4 (2017), 535–544. DOI: https://doi.org/10.1631/FITEE.1601232

[51] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. 2021. A review on the attention mechanism of deep learning. *Neurocomputing* 452 (2021), 48–62. DOI: https://doi.org/10.1016/j.neucom.2021.03.091

[52] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based fault localization. *Softw. Test., Verif. Reliab.* 25, 5-7 (2015), 605–628. DOI: https://doi.org/10.1002/stvr.1509 arXiv: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1509

[53] Christos H. Papadimitriou and John N. Tsitsiklis. 1987. The complexity of Markov decision processes. *Math. Oper. Res.* 12, 3 (1987), 441–450. DOI: https://doi.org/10.1287/moor.12.3.441

[54] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph K. Liu, and Robin Doss. 2019. Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey. *CoRR* abs/1908.08605 (2019).

[55] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the International Symposium on Software Testing*

and Analysis (ISSTA'15). Association for Computing Machinery, New York, NY, 24–36. DOI: https://doi.org/10.1145/2771783.2771791

[56] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of buggy code. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering (ICSE'16)*. 428–439. DOI: https://doi.org/10.1145/2884781.2884848

[57] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2021. EVMPatch: Timely and automated patching of ethereum smart contracts. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*. USENIX Association, 1289–1306. Retrieved from https://www.usenix.org/conference/usenixsecurity21/presentation/rodler

[58] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE'09)*. Association for Computing Machinery, New York, NY, 297–298. DOI: https://doi.org/10.1145/1595696.1595750

[59] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2021. Elysium: Context-aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts. DOI: https://doi.org/10.48550/ARXIV.2108.10071

[60] 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 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*. Association for Computing Machinery, New York, NY, 67–82. DOI: https://doi.org/10.1145/3243734.3243780

[61] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. Association for Computing Machinery, New York, NY, 269–280. DOI: https://doi.org/10.1145/2635868.2635875

[62] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. 25–36. DOI: https://doi.org/10.1109/ICSE.2019.00021

[63] Werner Vach. 2005. The dependence of Cohen's Kappa on the prevalence does not matter. *J. Clin. Epidem.* 58, 7 (2005), 655–661. DOI: https://doi.org/10.1016/j.jclinepi.2004.02.021

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. Retrieved from https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[65] M. Vladimirov and D. Khovratovich. 2018. ERC20 API: An Attack Vector on Approve/Transfer from Methods. (2018).

[66] Bin Wang, Han Liu, Chao Liu, Zhiqiang Yang, Qian Ren, Huixuan Zheng, and Hong Lei. 2021. BLOCKEYE: Hunting for defi attacks on blockchain. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. 17–20. DOI: https://doi.org/10.1109/ICSE-Companion52605.2021.00025

[67] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2021. Automated patch correctness assessment: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. Association for Computing Machinery, New York, NY, 968–980. DOI: https://doi.org/10.1145/3324884.3416590

[68] Xingya Wang, Haoran Wu, Weisong Sun, and Yuan Zhao. 2019. Towards generating cost-effective test-suite for ethereum smart contract. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*. 549–553. DOI: https://doi.org/10.1109/SANER.2019.8668020

[69] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. 1–11. DOI: https://doi.org/10.1145/3180155.3180233

[70] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime ClÃĬment, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Trans. Softw. Eng.* 43, 1 (2017), 34–55. DOI: https://doi.org/10.1109/TSE.2016.2560811

[71] Yatao Yang, Zibin Zheng, Xiangdong Niu, Mingdong Tang, Yutong Lu, and Xiangke Liao. 2021. A location-based factorization machine model for web service QoS prediction. *IEEE Trans. Serv. Comput.* 14, 5 (2021), 1264–1277. DOI: https://doi.org/10.1109/TSC.2018.2876532

[72] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Marina Meila and Tong Zhang (Eds.), Vol. 139. PMLR, 11941–11952. Retrieved from https://proceedings.mlr.press/v139/yasunaga21a.html

[73] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*. DOI: https://doi.org/10.48550/ARXIV.2203.12755

[74] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. Association for Computing Machinery, New York, NY, 1506–1518. DOI: https://doi.org/10.1145/3510003.3510222

[75] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 27 (Sep. 2020), 32 pages. DOI: https://doi.org/10.1145/3402450

[76] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2019. Predictive mutation testing. *IEEE Trans. Softw. Eng.* 45, 9 (2019), 898–918. DOI: https://doi.org/10.1109/TSE.2018.2809496

[77] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic smart contract protection made easy. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*. 23–34. DOI: https://doi.org/10.1109/SANER48275.2020.9054825

[78] Xiangfu Zhao, Zhongyu Chen, Xin Chen, Yanxia Wang, and Changbing Tang. 2017. The Dao attack paradoxes in propositional logic. In *Proceedings of the 4th International Conference on Systems and Informatics (ICSAI'17)*. 1743–1746. DOI: https://doi.org/10.1109/ICSAI.2017.8248566

[79] Zibin Zheng, Xiaoli Li, Mingdong Tang, Fenfang Xie, and Michael R. Lyu. 2022. Web service QoS prediction via collaborative filtering: A survey. *IEEE Trans. Serv. Comput.* 15, 4 (2022), 2455–2472. DOI: https://doi.org/10.1109/TSC.2020.2995571

[80] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. 2020. Automatic detection and repair recommendation of directive defects in Java API documentation. *IEEE Trans. Softw. Eng.* 46, 9 (2020), 1004–1023. DOI: https://doi.org/10.1109/TSE.2018.2872971

[81] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. *A Syntax-guided Edit Decoder for Neural Program Repair*. Association for Computing Machinery, New York, NY, 341–353. DOI: https://doi.org/10.1145/3468264.3468544

[82] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2021. Smart contract development: Challenges and opportunities. *IEEE Trans. Softw. Eng.* 47, 10 (2021), 2084–2106. DOI: https://doi.org/10.1109/TSE.2019.2942301

Received 6 March 2023; revised 18 July 2023; accepted 1 December 2023