

Security Code Recommendations for Smart Contract

Xiaocong Zhou¹, Yingye Chen¹, Hanyang Guo², Xiangping Chen³, Yuan Huang^{2,*}

¹School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China

²School of Software Engineering, Sun Yat-sen University, Zhuhai, China

³School of Communication and Design, Sun Yat-sen University, Guangzhou, China

{isszxc,chenxp8,huangyuan5}@mail.sysu.edu.cn, {chenyy287,guohy36}@mail2.sysu.edu.cn

Abstract—A smart contract is a self-executing program that is stored on the blockchain and runs when predetermined conditions are satisfied. Many frequent transactions involving asset transfers rely on smart contracts deployed on the blockchain, making them highly vulnerable to attack, thus it is essential to ensure the security of smart contracts. Since the smart contract is immutable once deployed, developers must try their best to fix existing vulnerabilities in advance to ensure security. Current approaches for automatic program repair on the smart contracts have mainly adopted the heuristic search algorithms or defined patterns to fix several well-defined types of vulnerabilities. They can only provide security code recommendations for developers in specific scenarios. We explore more general automated program repair of smart contracts in software history.

To pave the way for studying code changes related to bug fix of smart contracts in software history, we present a labeled public dataset for method-level program repair task, containing over 12 typical insecure code patterns. Unlike bugs in traditional software, the vulnerabilities of smart contracts are more associated with access control and conditional statements as smart contracts pertain to financial assets. For this problem, we devise a novel double-encoder network and use a code representation designed for the smart contract based on syntax information to repair program. By implementing and evaluating our approach on new dataset comprised of over 10,000 program pairs, we demonstrate the superiority of our approach in both qualitative and quantitative aspects.

Index Terms—program repair, smart contract, empirical software engineering

I. INTRODUCTION

Smart contracts are the computer programs running on a blockchain, which operate immutable logic without the need of an intermediary. They are developed using the high-level programming languages (PLs) (e.g., Solidity) and compiled into the low-level bytecode to be executed on blockchain platforms (e.g., Ethereum). Smart contracts on Ethereum are capable of holding large amounts of tokens, and many frequent transactions involving asset transfers rely on the execution of smart contracts deployed on Ethereum. This attracts attackers who hope to profit from exploiting security vulnerabilities and unexpected behaviors in smart contracts. Smart contract security incidents [1], [2] occur often and have led to significant financial losses.

To avoid the vulnerabilities in smart contract programming, related research has proposed a number of vulnerability detection tools that base on dynamic or static analysis, such as fuzzing techniques [3], [4], and symbolic execution [5],

[6]. These tools can detect vulnerabilities, but cannot provide security code recommendations or repair them automatically. With only vulnerability detection tool, developers need to manually fix the detected vulnerabilities through their own expertise. Research [7] has shown that developers almost spend half of the time in software development working on program repair. Besides, due to insufficient expertise on vulnerabilities, developers may introduce new security problems while repairing them [8]. In particular, unlike software projects developed in traditional PLs and deployed on online servers, smart contracts cannot be changed once deployed on a blockchain. Therefore, to improve the efficiency and ensure the security in the smart contract development, automated repair techniques or security code recommenders is helpful.

For smart contract PL Solidity, the approaches in existing automated repair tools are mainly based on heuristic search algorithms [9] or defined pattern-based methods [10]. Focusing on several common and important vulnerabilities, they explore automated program repair for smart contracts. However, as smart contracts are applied in more complex and diverse scenarios, the number of types of vulnerabilities they target is limited. Research [11] shows that there are at least 14 types of vulnerabilities in the development process and at least 13 types of vulnerabilities are introduced during the code review process. That means the security code recommendations can only be provided to developers in certain specific situations by existing automated repair techniques of smart contracts.

To address this issue, we explore more general automated program repair of smart contracts based on statistical learning, to better assist the developers in a variety of scenarios. The large availability of data, particularly source code, and its surrounding artifacts in open source repositories provides an objective condition for our study of software history. From 2,000 smart contract open-source projects, we extracted 24,153 method pairs related to program repair based on the commit message information. Each of them contains the method before and after bug-related commit. Among them, there are 12,219 method pairs in which the scope of the code change is on one-line, accounting for more than 50% of the total. Therefore, considering the data size and model capabilities, we focus on dataset composed of more than 10,000 method pairs related to program repair with one-line patches of smart contracts. This is by far the largest labeled dataset on program repair for smart contracts released publicly to our knowledge.

As smart contracts have been an important carrier of decen-

*Corresponding author.

```

contract ERC20 is IERC20, IERC20Metadata {
    ...
    function transfer(address from, address to,
        uint256 amount) internal {
        require(from != address(0));
        require(to != address(0));
        beforeTokenTransfer(from, to, amount);
        uint256 fromBalance = balances[from];
        require(fromBalance >= amount);
        unchecked {
            balances[from] = fromBalance - amount;
        }
        balances[to] += amount;
        emit Transfer(from, to, amount);
        afterTokenTransfer(from, to, amount);
    }
    ...
}

```

Fig. 1: ERC20 token contract example

```

library SafeMath {
    ...
    function sub(uint256 x, uint256 y) internal
        returns (uint256) {
        assert(x >= y);
        uint256 z = x - y;
        return z;
    }
    function mul(uint256 x, uint256 y) internal
        returns (uint256) {
        uint256 z = x * y;
        assert(x == 0 || z / x == y);
        return z;
    }
    ...
}

```

Fig. 2: SafeMath library example

tralised finance (DeFi) [12], most of them involve the storage and transfer of financial assets. So vulnerabilities in insecure smart contracts mainly arise from access control or conditional judgments, which may be different from the program written in the traditional languages (e.g. Java, C, etc.). These statements usually appear on the line above or below a crucial operation such as an asset transaction in smart contracts. Fig. 1 and 2 show the statements (text highlighted in orange red) related to conditional judgments and access control in two of the most reused subcontracts [13] in development of smart contracts. In the vulnerable smart contracts, large number of vulnerabilities are associated with the context of the buggy lines, i.e., these statements related to conditional judgments and access control. For example, one of the most serious vulnerabilities in history of smart contracts, the Reentrancy vulnerability, is that the state change occurs in the next line of the buggy line (i.e., *call.value* function). And program repair for this vulnerability just requires moving the next line of the buggy line to

the previous line. So in order to generate one-line patches effectively, the contextual information of the buggy line needs to be fully considered.

In order to address this issue, we propose an end-to-end approach for smart contract repair with one-line patches, called SMARTREP. To prevent the vocabulary explosion problem while still retaining the maximum amount of syntactic information of the source code, we abstract the entire method. For low frequency words in the source code, only their syntactic types and relative positions are fed into the neural network. And as the contextual information of the buggy line mentioned above, we extract three-line buggy code in the buggy method. Next, we adopt an double-encoder structure to encode the abstracted method and three-line buggy code snippet respectively and integrate information with attention mechanisms.

We combine the hidden layer information as the input of decoder, and concretize the corresponding output into real code. We use beam search to generate multiple alternatives for code editing recommendations. Subsequently, we discard ungrammatical methods, and the rest of the scenarios are presented and evaluated.

We demonstrate a quantitative and qualitative evaluation of our approach SMARTREP respectively. For quantitative analysis, We implement SMARTREP in our dataset and evaluate it on 2,248 real-world buggy methods. Our best trained model is able to completely repair 535/2,248 (23.8%) testing samples. As a comparison, we implemented the model of the state-of-the-art approaches and the other related work on our dataset. Experiments show that, SMARTREP outperforms the other approaches in terms of completely repair rate of insecure smart contracts. For qualitative analysis, compared to existing automatic program repair approaches for smart contracts, SMARTREP is able to fix more types of vulnerabilities. Code editing operations for over 13 kinds of vulnerabilities can be recommended for developers of smart contracts. The implementation of SMARTREP and our dataset are available at <https://github.com/AnonymousGithub5/SmartRep>.

This paper is structured as follows. Section 2 presents 12 types of vulnerabilities commonly in insecure smart contracts. Section 3 describes our approach SMARTREP for secure smart contract recommendations. Section 4 evaluates SMARTREP on over 2000 real-world buggy smart contracts from open-source projects. Section 5 presents some Solidity-specific cases of program repair. Section 6 talks about related work, separately for smart contracts and for other popular PLs. Section 7 concludes the paper.

II. BACKGROUND

We refer to the description of vulnerabilities in smart contracts from: (1) five literature reviews [14]–[18] about smart contract vulnerability, (2) an implementation of the weakness classification scheme, Smart Contract Weakness Classification (SWC) Registry [19] and (3) a community-developed list of software and hardware weakness types, Common Weakness

Enumeration (CWE) [20]. Below we describe 11 types of the vulnerabilities that appear in our dataset.

Use of Deprecated Functions. Solidity is a relatively new language, which is evolving rapidly. Some APIs have been deprecated, or will be updated in the future, such as *throw*, *callcode* and *msg.gas*. These APIs may still be supported by the current version of the compiler. However, continued use of them in newer compiler versions may produce some unknown errors.

Unchecked Return Value. For the low-level function calls such as *send*, Ethereum Virtual Machine does not pass the status of exceptions thrown in function calls along call stack, but only indicates in the return value. When these function calls fail due to network errors or insufficient gas, a Boolean value *False* will be returned, but no exception will be thrown.

Incorrect Inheritance Order. Solidity language allows for multiple inheritance. A derived contract may inherit from several base contracts. Defining the same function in two or more base contracts can lead to ambiguity in inheritance. Solidity implements C3-algorithm to handles this ambiguity, which assigns a priority to each base contract. Therefore, it is unsafe to ignore the inheritance order because it can lead to unexpected behavior.

Illegal Coverage. In Solidity language, variables in functions without type declarations would be of *memory* type by default, and only take effect when the function is called, and will be recycled when returns. However, some versions of compilers may treat the default declaration of an array or structure type variable as a *storage* type. As a result, operations on these variables will result in illegal overwrites of storage, which may cause the contract to run incorrectly.

Transaction Ordering Dependence. Transaction Ordering Dependence is a vulnerability caused by program logic in smart contracts that uses only the order of transactions as a decision condition. A smart contract is invoked by initiating a transaction, but there is no strict correlation between when the transaction is initiated and when it is confirmed so that the contract invocation takes effect.

Reentrancy. Modifying a storage variable and transferring it in a smart contract ought to be an atomic operation. A reentrancy vulnerability may occur when the smart contract transfer the ether to an untrusted address and subsequently modify storage variables. If the target of the transfer is a contract with a malicious fallback function, it is possible for the malicious contract to launch a recursive call to the victim contract. Thus, the atomicity of the operation can be broken and the check can be bypassed to repeatedly gain the transfer.

Erroneous Visibility. This vulnerability appears when the visibility of a function is improperly defined, thereby allowing unauthorized access. Solidity language has provided four types of visibility to limit accessibility to functions of a contract, that is, *public*, *external*, *internal*, and *private*, indicating respectively that a function may be called arbitrarily, externally only, internally only, or only inside the contract. Functions that are not allowed to be called from external contracts should be defined as *private* or *internal*.

Arithmetic Issue. Solidity language have a specific length limit on the storage space for integer types, and therefore have a size limit on the integer types they can represent. Once the result of an integer operation exceeds this range, an integer overflow occurs. The arithmetic issue in smart contracts is mainly caused by the users who do not safely detect overflow or underflow on the result of operations.

Missing Return Statement. There are functions that indicate a return value, but not return anything. For such contracts, Ethereum virtual machine adds a default return value when it compiles the source code to low-level bytecode. They might adopt the return value to deal with the execution of the code and cause unpredictable errors because the caller might not be aware of the source code of the callee contract.

Authorization Through tx.origin. *tx.origin* is a global variable in Solidity language that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authorization may cause a vulnerability if an authorized account invokes a malicious contract, since an attacker can skip the authentication phase by using this property of *tx.origin*.

Erroneous Variable Type. Ethereum smart contracts are written in the high-level language Solidity, which is a strongly typed programming language that automatically checks for type matching errors in programs, such as when assigning a string to an integer variable. Neglecting to manually check for data types can easily lead to unexpected vulnerabilities.

III. APPROACH

Our approach starts with collecting bug-related commits from the 2,000 relevant repositories on GitHub. Based on the commit message information, we extract the source code before and after the commits, where each pair serves as an instance of an artificial program repair. Regarding one-line patch, we define the buggy line, extract the buggy context, and abstract the buggy method in the method pair of the program repair. To simulate the process of program repair, we feed the abstracted method and the extracted context into our model respectively to train the neural network. Then the inference results of the neural network is concretized into real patches. Besides, we provide multiple security code recommendations using beam search algorithm and the syntax parser.

A. Data Collection

At first, we use the GitHub ReST APIs¹ to download the top 2,000 Git repositories related to smart contracts from GitHub, ranked by the GitHub Stars², 1,000 of which are mainly in Solidity, and the rest in JavaScript. With the APIs, we retrieve the list of all commits submitted for review for each project. To remove commits not related to bugs, we follow previous work [21] and keep only commits that have the keywords “fix,” “bug,” “resolve,” or “patch” in their commit messages. We also filter commits using six commit messages anti-patterns: “rename,” “clean up,” “refactor,” “merge,” “misspelling,” and

¹<https://docs.github.com/en/rest>

²<https://stars.github.com>

“compiler warning.” Afterwards, the GitHub APIs is used to retrieve the pre-commit version and the post-commit version of the files in these project. We obtained 25,427 file pairs, each of them representing the file before and after bug-related commit.

Next, we extract the method from each pair of files using a Solidity parser. We adopt the method-level granularity for a couple of reasons: (1) methods implement an individual functionality and have adequate contextual information to fix the vulnerabilities inside; (2) empirical studies [22] shows that small scope is a feature of an individual program repair on in software development; (3) file-level and contract-level code changes are still possible by composing multiple method-level code transformations; (4) files and contracts as the object of study will generate plenty of noise, which affects the extraction and analysis of vulnerability information and hinder our goal to train an automatic program repair model.

With the file name, contract name and method name, we create a method-to-method mapping. We only select method pairs where the post-commit code has changed by less than two lines compared to the pre-commit code. Since we focus on one-line patches, we do not consider line deletion in code change. Finally, the qualified methods are represented as a list of paired method $\{(m_b, m_a)_1, (m_b, m_a)_2, \dots, (m_b, m_a)_n\}$, where each pair $(m_b, m_a)_i$ contains the method before the commit (m_b) and the method after the commit (m_a). This resulted in 12,231 method pairs, each of them representing the method before and after bug-related commit. These are examples of method pair used to train a sequence-to-sequence model to translate m_b in m_a .

B. Data Processing

When performing method abstraction and obtaining enough information of vulnerabilities in smart contracts, there are two challenges: (1) The source code contains a large number of uncommon tokens such as account addresses, asset values, etc., which can cause large corpus of text when fed directly into the neural network. When training model based statistical learning, overly large vocabularies result in more parameters, decreased accuracy, and increased time. (2) Most of the vulnerabilities are associated with the context of the buggy lines in smart contracts, especially statements related to conditional judgments and access control. To generate one-line patches efficiently, these critical pieces of information ought to be captured.

Buggy Line. As mentioned above, we are targeting one-line patches. The code editing operations for program repair in our dataset could be mainly classified into two types: (1) changing one line in the method, and (2) adding one line before another line in the method. For the first type, we define the line of code being changed as the buggy line. And for the second type, we define the next line of the added line as the buggy line. By manipulating the buggy line, that is, modifying the tokens inside or inserting the fixed line, we can fix the vulnerability in the insecure smart contracts. We define the buggy line in

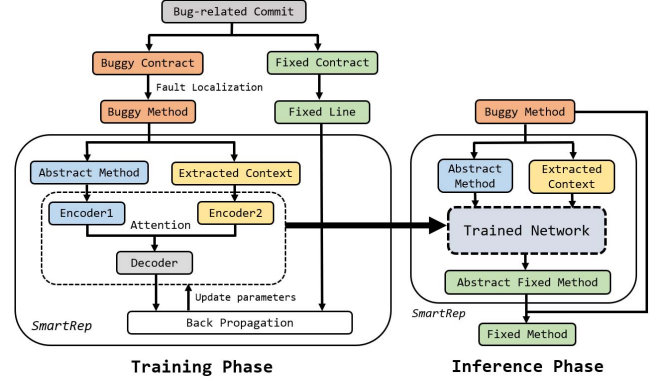


Fig. 3: Overview of SMARTREP for program repair

order to limit the scope of the buggy contextual information extraction.

Extracted Context. Many vulnerabilities in smart contracts are closely associated with the statements related to access control and conditional judgments. The declarations of these statements often appear in buggy method not far from the buggy line, especially on the line before or after. It is not enough to focus only on the information in the buggy line. We extracted three lines in the buggy method as the buggy context, which are the previous line of the buggy line, the buggy line, and the next line of the buggy line. These three-line buggy code snippet usually contain the key information of the buggy line, such as reckless access validation which causes unauthorized manipulation, and lack of checking the maximum value after an increment operation.

Abstract Method. When applying neural language models to source code, large vocabularies and out-of-vocabulary issues are always major challenges. Referring to the previous work [23], to address this problem, we propose a new code representation for smart contracts based on abstract syntax tree to reduce the vocabulary. First, we compute the top-400 most frequent tokens in the source code, which we want to preserve in our vocabulary. Next, we feed the source code into a Solidity lexer, and then get a sequence of tokens. Given the sequence of tokens of a method, we generate an abstract syntax tree (AST) with a Solidity parser using ANTLR³ grammar. Then we use type information of the AST, that is, identifiers (e.g. modifier invocation, assembly assignment) or literals (e.g. decimal number, boolean literal), and a numerical ID, which is generated sequentially in current buggy method. For example, we represent the definition of the first function call in buggy method as `FunctionCall_0`, and the second identifier as `Identifier_1`. The code representation of the smart contract reduces the vocabulary while preserving the maximum amount of syntactic information. We feed the abstract method into the model as one of the inputs to get the information of the entire buggy method. Besides, we store the mapping from the replaced representation to the source

³<https://github.com/solidity-parser/antlr>

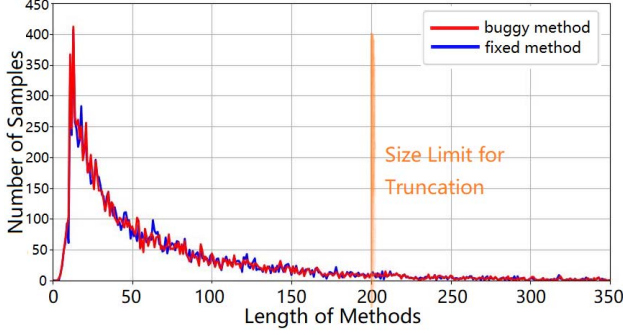


Fig. 4: Only 8% of samples exceed the 200 tokens length

information of the low-frequency words in a dictionary, each corresponding to a method, for the reduction of source code after the generation of the fixed abstract method.

C. Statistical Learning

As one of the most common frameworks in deep learning, encoder-decoder model architecture implements conversion from source text to target text, which is used in machine translation, text simplification, document summarization, etc. Encoder takes a variable-length sequence as input and outputs a constant-length vector, and decoder converts the vector into a variable length output.

Automatic program repair with encoder-decoder architecture is similar to neural machine translation, which translates one language into another. The neural network learns to translate buggy program into fixed program. We adopt a double-encoder network to generate one-line patches for program repair in insecure smart contracts. The first encoder is designed for the abstract method, aiming to retrieve information about the entire method. And the second is for the extracted context, which focuses on capturing information about access control, conditional judgment statements and the buggy line. Besides, attention mechanism to address the issue of information loss due to excessive message length. When inferring a token, the attention mechanism allows the model to focus only on the most relevant tokens in the original text.

We build a dictionary for the source code, and convert each token in the source code into a one-hot vector. We feed the source one-hot vector sequence of abstract method $x = [x_1, x_2, \dots, x_n]$, where x_i represents an one-hot vector, and n means the token length of the abstract method. Then x is fed into the embedding layer:

$$v_{m,t} = w_m x_t \quad (1)$$

where x_t is the one-hot vector at time t , w_m is the weight matrix in the embedding layer, and v_t is embedding vector of the source token at time t . We input the embedding vector at time t into the long short-term memory (LSTM) module and then get the output of the corresponding hidden layer:

$$h_t, c_t = \text{LSTM}(h_{t-1}, c_{t-1}, v_{t-1}) \quad (2)$$

where h_t is the hidden state at time t , c_t is the cell state at time t and v_m is the embedding matrix of the source token. To better capture the input information, we introduce the attention mechanism. For the first encoder, we compute attention weight $a_{m,j}$ at each time j between s_j and the hidden layer h_i at each time i :

$$a_{m,j} = \sum a_{i,j} h_i \quad (3)$$

where $a_{i,j}$ is defined as follows:

$$a_{i,j} = \sigma \{ W_1^T \cdot \tanh [W_2 \cdot \text{cat}(h_i, s_j)] \} \quad (4)$$

where s_j is the output of the decoder at time j , h_i is the hidden state at time i . W_1^T is a parameter vector and W_2 is a parameter matrix of the neural network. Function cat is to concatenate the given sequences in the given dimension. We treat the input of the second encoder in the same way. Then we obtain $a_{ct,j}$, the attention weight at each time j for the second encoder. We merge the messages of encoder1 and 2 together as one of the inputs to the decoder:

$$a_j = a_{m,j} + a_{ct,j}. \quad (5)$$

Subsequently, we aim to get the fixed line as the output of the decoder. The decoder has also a LSTM recurrent neural network and turns the fixed line into the embedding vector as first. Then it integrates the information from encoders and the input from decoder to obtain inference as follows:

$$s_{j+1} = \tanh [b + W_3 \cdot \text{cat}(s_j, a_j, v_{d,j+1})] \quad (6)$$

where b and W_3 are parameter matrices of the neural network, $v_{d,j+1}$ is the embedding vector of the decoder's input at time $j+1$, s_j is the output of the decoder at time j and a_j is the weighted average attention weights from two encoders at time j .

D. Patch Generation

The dictionaries mentioned in Sec. III-B map the concrete identifiers and literals to abstract tokens, so they can be used to reduce abstract tokens to concrete code. After obtaining the output of the decoder, we concretize the output of the decoder into real code as the patch. Then we apply the generated patch to the buggy method, that is, insert the one-line patch as the previous line of the buggy line or replace the buggy line with the new one. Finally we can get a complete fixed method.

Besides, we use beam search, a heuristic search algorithm used in many neural language models as a final layer, to select several best scenarios. Among the different options generated by beam search algorithm, we filter out the ungrammatical methods by a Solidity parser and recommend the rest to smart contract developers.

IV. EVALUATION

In this section, We evaluate our approach experimentally.

A. Research Questions

RQ1: How does the performance of SMARTREP compare to statistical learning-based repair technologies for traditional PL in generating correct one-line patches?

TABLE I: Amount of method pairs and vocabulary

	<i>Amount</i>	V_{before}	V_{after}
Small	6,170	15,002	445
Medium	4,936	42,250	977
Total	11,244	53,591	992

RQ2: To what extent does the second encoder improve the efficiency of automatic program repair? Is extracting three-line buggy code snippet as extracted buggy context the best choice for program repair of smart contracts?

RQ3: What types of vulnerabilities can we fix? How effective is our method compared to other smart contract automated program repair tools?

B. Experimental Methodology

Programming. We adopt Python as the programming language, and PyTorch⁴ platform to build the deep learning framework for model construction.

Dataset. In Fig. 4, we find that most of the methods in the method pairs are within 200 tokens. Therefore, We abandoned the method pairs that contain methods exceeding this limit. To verify the ability of SMARTREP to fix buggy code with different lengths, we set up three datasets, which are **Small**, **Medium**, and **Total**. The dataset **Small** contains only methods with less than 50 tokens, while the dataset **Medium** contains only methods with more than 50 tokens and less than 200 tokens. And the dataset **Total** is contains the methods with less than 200 tokens. The exact number of method pairs in each dataset is shown in Table I.

Vocabulary. Through the abstraction process mentioned in Sec. III-B, the size of vocabulary of dataset **Small** is reduced from 15,002 (V_{before}) to 445 (V_{after}), and that of dataset **Medium** is reduced from 42,250 to 977. Also, that of dataset **Total** is reduced from 53,591 to 992 as shown in Table I.

Parameters. Each encoder consists of an embedding layer and two LSTM layers, and the decoder consists of an embedding layer and a LSTM layer.

- *Embedding layer.* The size of the dictionary of embeddings is 1024, and the size of each embedding vector is 256.
- *LSTM layer.* The number of expected features in the input is 256, and the number of features in the hidden state is 256 as well. When the length of input is less than 50, the number of recurrent layers is 3, otherwise it is 4.
- *Attention mechanism.* There are 4 linear layers used to consist of the attention mechanism. The size of each input sample of the linear layers is 256, and the size of each output sample is 256 as well.

Loss Function. The bilingual evaluation understudy (BLEU) [25] is a metric for evaluating the generated statements, which is the predicted result of an automatic machine translation

system. The Natural Language Toolkit (NLTK) library⁵ written in Python provides an implementation of BLEU scoring, and we use it to evaluate the generated method and calculate loss value during the training in our approach by comparing it to the corresponding fixed method in our dataset.

1) *Methodology for RQ1:* With the configuration above, we experiment on our dataset. Our dataset is divided into a training set and a testing set at a ratio of eight to two. To the best of our knowledge, among the program repair approach for one-line patch, SequenceR [24] is the state-of-the-art approach. SequenceR was compared with the work of Tufano et al. [23] when it was proposed, so we compare them both. Besides, following previous work [24], we train a simple sequence-to-sequence network to translate the buggy line into the fixed line, as one of the baselines. We use two metrics, *structurally repair* and *completely repair*, to measure the effectiveness of the smart contract repair. They are defined as follows:

Definition 1. *structurally repair* refers that the output of the model is the same as the abstract fixed method. In other words, the model can infer the structure of the fixed method, i.e., it is able to predict the type information of each token but not necessarily the specific name of the identifiers or the value of the literals in the fixed method.

Definition 2. *completely repair* refers that the inference result of the model is the same as the labeled fixed method after the code concretization phase.

2) *Methodology for RQ2:* For the second encoder, we performed ablation experiments to evaluate the effect of different buggy context on generating patches. As mentioned above, the input of the second encoder aims to capture the contextual information of the buggy line associated with the access control and conditional judgment statements. Table III shows the four experiments we perform as follows: (1) we remove the second encoder and just feed the abstract buggy method into the model; we input (2) the buggy line and the previous line of it, (3) the buggy line and the next line of it, and (4) the buggy line, the previous two lines and the next two lines, to the second encoder. Under this setting, the first experiment demonstrates the effect of the buggy context, and the latter three experiments explore the best way to extract the contextual information.

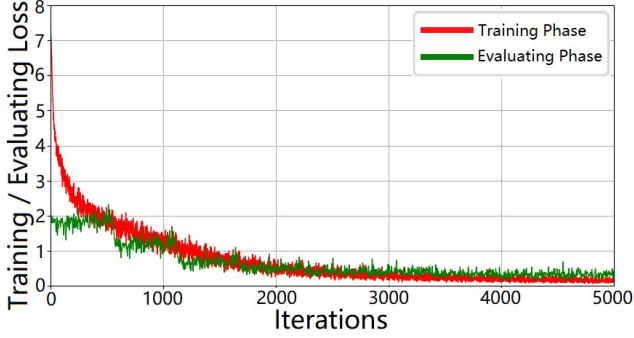
3) *Methodology for RQ3:* We analyse 11 typical vulnerabilities that appear in our dataset, which are described in detail in relevant literature reviews [14]–[18], SWC [19] and CWE [20]. We counted the number of each type of vulnerability separately in our dataset. Based on the statistical results, we evaluated the performance of SMARTREP on the testing set for fixing the vulnerabilities of these 11 types. Besides, to our best knowledge, the approaches propose in [9] and [10] are the two most effective tools for smart contract program repair, using a genetic algorithm and a defined pattern-based approach respectively. Unfortunately, the approach proposed in [10] does not provide open source code, so we cannot do a

⁴<https://pytorch.org>

⁵<https://github.com/nltk/nltk>

TABLE II: Comparison with statistical learning-based repair technologies for traditional programming languages

	Small		Medium		Total	
	<i>structurally</i>	<i>completely</i>	<i>structurally</i>	<i>completely</i>	<i>structurally</i>	<i>completely</i>
<i>seq2seq(line2line)</i>	135/1,234 (10.1%)	97/1,234 (8.0%)	45/992 (4.5%)	24/992 (2.4%)	215/2,248 (9.6%)	150/2,248 (6.7%)
<i>Tufano et al. [23]</i>	281/1,234 (22.8%)	162/1,234 (13.1%)	187/992 (18.9%)	86/992 (8.7%)	407/2,248 (18.1%)	213/2,248 (9.5%)
<i>SequenceR [24]</i>	390/1,234 (31.6%)	264/1,234 (21.4%)	316/992 (31.8%)	141/992 (14.2%)	769/2,248 (34.2%)	487/2,248 (21.6%)
SMARTREP	464/1,234 (37.6%)	304/1,234 (24.6%)	275/992 (27.7%)	147/992 (14.8%)	966/2,248 (42.9%)	535/2,248 (23.8%)

**Fig. 5: Training and evaluating loss change on dataset Total**

comparison. We use a marker "✓" in Table IV to indicate the types of vulnerability the approach can fix. Meanwhile, the approach in [9] is designed based on the deployed contracts on Etherscan⁶, and uses the test case obtained by simulated replay on the blockchain as one of the fitness functions. But our data is collected from GitHub, most of them have not been deployed on the blockchain. Therefore, we implement it using the number of targeted vulnerability and compilation results as the fitness functions in genetic algorithm, and use the repair result on our dataset as baseline. We set the generation size in this genetic algorithm to 10 for each sample. The five mutations with the best performance of fitness function in the last generation are selected as repair result. Our experimental results are in the Table IV, where the introduction of the vulnerabilities can be found in Sec. II, and the specific examples are in Sec. V.

C. Experimental Results

1) *Answer to RQ1:* We implement SMARTREP on the three datasets Small, Medium and Total respectively, and the loss change is shown in Fig. 5.

As shown in Table II, SMARTREP outperforms the state-of-the-art approach [24] for one-line patches on program repair in terms of inference of correct code structure and correctness of repair. SMARTREP is able to fix 966/2,248 (42.9%) vulnerabilities structurally and 535/2,248 (23.8%) vulnerabilities completely on the dataset Total, while SequenceR fixes 769/2,248 (34.2%) vulnerabilities structurally and 487/2,248 (21.6%) vulnerabilities completely. The approach [23] and

⁶<https://etherscan.io>

TABLE III: Ablation experiments for extracted context

Input to the Second Encoder	Total
Nothing (remove the second encoder)	300/2,248 (13.3%)
Previous line & Buggy line	488/2,248 (21.7%)
Buggy line & Next line	507/2,248 (22.5%)
Two previous line & Buggy line & Two next line	387/2,248 (17.2%)
Previous line & Buggy line & Next line (SMARTREP)	535/2,248 (23.8%)

the line-to-line model fix fewer vulnerabilities structurally and completely than theirs. Similarly, SMARTREP can fix more vulnerabilities on datasets Small and Medium, which illustrates that SMARTREP is adaptable to different lengths of input well. In comparison with the approaches proposed in related work, we demonstrate the effectiveness of abstract method and extracted context in insecure smart contracts. We consider that SequenceR treats the code other than the buggy line in the buggy method equally and does not pay special attention to the context near the buggy line. And the double-encoder network is able to capture useful contextual information related to vulnerabilities and significantly improve the performance of generating one-line patches.

2) *Answer to RQ2:* To understand the usefulness of the contextual information captured by the second encoder to patch generation, we performed an ablation experiment specifically on the second encoder. As shown in the Table III, in the first ablation experiment, the model fixes 300/2,248 (13.3%) vulnerabilities in the testing set, which is decreased by 10.5% compared to SMARTREP. This decrease illustrates that the buggy context captured by the second encoder plays an important auxiliary role in patch generation. In order to explore whether three lines are the best context extraction, we changed the scope of the context extract in the latter three experiments. Experiments show that narrowing the scope of the extract context resulted in 2.1% and 1.9% decrease in the completely repair rate, while expanding the scope also caused a 6.6% decrease, as shown in Table III. We can infer that a smaller scope cannot cover enough statements about conditional judgments and access control, while a larger scope tends to introduce noise, making it difficult for the model to learn. The extraction of the previous line, the buggy line and the next line is most beneficial for the model to generate one-line patches for program repair.

TABLE IV: Comparison with two smart contract automated program repair tools

	[10]	SCRepair [9]	SMARTREP
Use of Deprecated Functions	-	-	65/101
Unchecked Return Value	✓	7/17	8/17
Incorrect Inheritance Order	-	-	12/55
Illegal Coverage	-	-	5/26
Transaction Order Dependence	-	0/3	0/3
Reentrancy	✓	2/21	6/21
Erroneous Visibility	-	-	79/214
Arithmetic Issue	✓	0/90	23/90
Missing Return Statement	-	-	1/18
Authorization Through tx.origin.	-	-	9/14
Erroneous Variable Type	-	-	48/111
Others	-	-	279/1578
Total	-	9/131	535/2248

3) *Answer to RQ3*: The results of our experiments are presented in Table IV. SCRepair [9] is designed for four types of vulnerabilities, which are **Transaction Order Dependence**, **Unchecked Return Value**, **Reentrancy** and **Arithmetic Issue**. So we only implement this approach on 131 vulnerabilities that belong to these four types in our dataset. Among them, SCRepair is able to fix 2/21 (9.5%) vulnerabilities of **Reentrancy** and 7/17 (41.2%) vulnerabilities of **Unchecked Return Value**, whereas there is no vulnerability of the other two types fixed by it successfully. And our approach SMARTREP outperforms SCRepair in fixing the vulnerabilities of these four types, and infers the correct patches for the vulnerabilities of the other six types. We consider the missing of test cases as a component of the fitness function may have an impact on the performance of SCRepair. This facilitates SMARTREP outperforms the heuristic-based approach in terms of success rate of program repair. Besides, SMARTREP also has advantages over the approach based on genetic algorithm in terms of training and inference time. Our test machine is an Intel Core i5-10980XE with 256 GB main memory, which has 18 physical cores and 36 virtual cores, and a NVIDIA GeForce RTX 3090 GPU. With our experimental setup, SCRepair took almost 27 days to run on the 131 contracts, while the training and validation process of SMARTREP required only a total of about three hours. This shows that providing the same number of contracts, SMARTREP is able to provide security code recommendations much more quickly.

In addition, for the vulnerabilities that appear in our dataset but are not described in our reference [14]–[20], we categorized them into type **Others** as shown in Table IV. It can be seen that SMARTREP can also repair several types of these vulnerabilities by learning the software history and gaining generalization ability. We also show some specific instances corresponding to the common types of these vulnerabilities in Sec. V.

V. CASE STUDY

We have observed that SMARTREP can fix multiple types of vulnerabilities in insecure smart contracts, and we present some successful cases in this section. Sec. V-A shows the cases of 10 types of vulnerabilities that have been reported in the five literature reviews [14]–[18] or included in the SWC [19] and CWE [20]. Their patterns and damage have been widely discussed, so we refer to them as typical vulnerabilities. Sec. V-B shows 3 cases of other vulnerabilities that are not widely discussed, but are commonly found in the development of smart contracts.

A. Typical Vulnerabilities

Type 1. Illegal Coverage

```
- function getTags() external view returns (
    bytes32) {
+ function getTags() external view returns (
    bytes32 memory ) {
```

Listing 1: *TestSTOFactory.getTags()*

Type 2. Reentrancy

```
- if(msg.sender.call.value(_am)()) { balances[
    msg.sender] -= _am;
+ if(msg.sender.send(_am)) { balances[msg.
    sender] -= _am;
```

Listing 2: *ETH_VAULT.CashOut()*

Type 3. Incorrect Inheritance Order

```
- function init(BaseWallet _wallet) public
    virtual onlyWallet(_wallet) {
+ function init(BaseWallet _wallet) public
    virtual override onlyWallet(_wallet) {
```

Listing 3: *BaseModule.init()*

Type 4. Authorization Through tx.origin

```
- require(tx.origin == owner);
+ require(msg.sender == owner);
```

Listing 4: *Ownable.addMeByRC()*

Type 5. Missing Return Statement

```
- function onTokenTransfer(address, uint,
    bytes calldata) external ;
+ function onTokenTransfer(address, uint,
    bytes calldata) external returns(bool);
```

Listing 5: *ITransferReceiver.onTokenTransfer()*

Type 6. Use of Deprecated Functions

```
- if (!approve(_spender, _amount)) throw;
+ if (!approve(_spender, _amount)) revert();
```

Listing 6: *HHO.approveAndCall()*

Type 7. Erroneous Visibility

```
- function insert(uint256 storage heapList,
  uint256 k) public {
+ function insert(uint256 storage heapList,
  uint256 k) internal {
```

Listing 7: *PriorityQueue.insert()*

Type 8. Unchecked Return Value

```
- honlninja.send(this.balance - 1 ether);
+ if (!honlninja.send(this.balance - 1 ether))
  revert();
```

Listing 8: *HumansOnlyNetworkETHpreICO.Partial8Send()*

Type 9. Arithmetic Issue

```
- balanceOf[to] += value;
+ balanceOf[to] += value; require(balanceOf[to]
  ] >= value);
```

Listing 9: *RENAULT_301201.transfer()*

Type 10. Erroneous Variable Type

```
- if(recipientImplementation != 0)
+ if(recipientImplementation != address(0))
```

Listing 10: *OrcaToken.callRecipient()*

B. Other Vulnerabilities

In this section, we list several common vulnerabilities and their fixes by SMARTREP in software history of smart contracts. We only display Solidity-specific program repair, as other general code changes are available in other related work [23], [24]. Although these types of the vulnerabilities have not been discussed in our reference [14]–[20], fixing them can also improve the security of smart contracts. This also demonstrates SMARTREP’s ability to provide developers with security code recommendations in multiple scenarios.

Modifier Addition. The modifier in Solidity language is similar with the aspect in Java Spring framework, which controls the logic of the function. The addition of modifier to a function is able to prevent some unexpected situations. For instance, functions that are not declared as *payable* cannot receive any Ethereum payments, and modifier *noReentrancy* locks the contract at execution time and unlocks it at the end of execution, which could prevent the Reentrancy vulnerability.

Type 11. Modifier Addition

```
- function makeBet() {
+ function makeBet() payable {
```

Listing 11: *Lottery.makeBet()*

Solidity Keyword Addition. There are a number of keywords specific to Solidity language, such as *exit*, *event*. If these words are omitted, there may be a lack of functionality. For example, if you do not declare an *exit* in a statement, the corresponding event will not be triggered.

Type 12. Solidity Keyword Addition

```
- LogSetAuthority(authority);
+ emit LogSetAuthority(authority);
```

Listing 12: *DSAuth.setAuthority()*

SafeMath Library Use. SafeMath Library is one of the most common libraries for safe arithmetic operations in Solidity language and is part of the open source smart contract development tool OpenZeppelin. The functions *add*, *sub*, *mul*, and *div* of SafeMath library do simple quadratic operations, and then throw an error when an overflow or underflow occurs. So using SafeMath library to develop smart contracts is able to prevent arithmetic issue.

Type 13. SafeMath Library Use

```
- return totalSupply - totalBorrow;
+ return totalSupply.sub(totalBorrow);
```

Listing 13: *LoanTokenLogicStandard.marketLiquidity()*

VI. RELATED WORK

We introduce the related work of automatic program repair technologies in Sec. VI-A and vulnerabilities detection and fix in insecure smart contracts in Sec. VI-B.

A. Automatic Program Repair

The field of automatic program repair in traditional languages (e.g. C++, Java) is mature and has well-defined techniques that can mainly be classified into approaches based on heuristic search, defined pattern, semantic analysis and statistical learning.

Heuristic search-based approaches like GenProg [26] and PraPR [27] apply mutations to the source code, and adopt random search or genetic algorithm to find plausible patches so that the mutated program can pass all failing test cases.

Defined pattern-based approaches refers to the process of predefining some patch templates [28] or patch generation strategies [29] to guide program repair, depending on the developers or researchers. **Semantic analysis-based** approaches such as SemFix [30], and SemGraft [31] first infer a required specification or a repair constraint of the insecure source code, which is usually achieved via symbolic execution of the test program. Then they synthesize patches for the buggy statements based on the inferred specification using program synthesis techniques.

Statistical learning-based approaches have also developed rapidly in recent years due to the rise of machine learning

and massive amounts of data. The program repair tool DeepFix[30] used a sequence-to-sequence neural network model to automatically fix compilation errors in C language. Research [23] explored the effectiveness of using neural machine translation (NMT) techniques in natural language processing to learn meaningful code change, which shows promising results towards automatic program repair based on machine learning. This approach adopts an encoder-decoder model, where the input and output of the model are the code before and after code review respectively, which targets a wider range of code change types and higher complexity. Basing on the work of it [23], SequenceR [24] introduces a copy mechanism to support automatic program repair for Java language. The scope of program repair of SequenceR is on an one-line patch, as we do. CoCoNuT [21] combines convolutional neural network into NMT model, to explicitly capture the contextual information of the buggy source code. The approaches using statistical learning hold the promise of correctly fixing real-world program vulnerabilities in large projects, with the advantage of not relying on any defined pattern as well as heuristic rules. There is still much room for improvement in the performance of statistical learning-based approaches to automated program repair, especially for end-to-end patch generation models. To the best of our knowledge, statistical learning is rarely used to automatically fix vulnerabilities in insecure smart contracts.

B. Security Analysis of Smart Contract

We introduce the work related to security analysis of smart contracts, including vulnerability detection and automatic program repair.

The vulnerability detection approaches for smart contracts use the techniques that can mainly be classified into symbolic execution [6], fuzzing [32] and formal verification [33]. Oyente [6] utilizes the bytecode of the smart contracts and the global state of Ethereum to build a control flow graph combined with symbolic execution for vulnerability identification. Contractfuzzer [32] generates test cases based on the ABI specification of the smart contracts, and then analyzes the runtime state of the contracts to identify the vulnerabilities. Securify [33] defines rules for security properties and determines a buggy contract if it violates the defined rules.

Existing automatic program repair tools in smart contracts mainly use heuristic algorithms [9] or defined pattern-based methods [10]. SCRepair [9] applies genetic algorithms to narrow the search space for candidate patches, while introducing the calculation of gas consumption to generate more efficient vulnerability patches for the buggy contracts. Heuristic search-based approaches have the advantage of being more general and applicable to different kinds of vulnerabilities, but they tend to generate plausible patches and affect the accuracy of inference. SMARTSHIELD [10] designs remediation rules at the bytecode level to fix the vulnerabilities by transforming the control flow and inserting data protection operations. The quality of the patches produced by the defined pattern-based approaches is improved compared to that of the heuristic search-based approaches. However, the repair capability and

application scope of these approaches are directly determined by the defined rules.

Besides, automatic program repair in smart contracts could also be seen as a specific kind of program update task. The approach proposed in [34] conducts an empirical study on the repeatability of smart contracts by using cluster analysis, and tries to extract the differentiated codes to guide the smart contract update. We also learn code changes from program history, but using a statistical learning-based approach allows us to learn more operations for vulnerability fixes and provides a wider variety of code recommendations.

VII. THREATS TO VALIDITY

Firstly, we only focus on Solidity because it is the most popular development language for smart contracts [35] in this paper. However, smart contracts can be built by other high-level PLs, such as the Vyper language. So designing automatic program repair tools for the smart contracts written in other PLs needs to consider extra language features.

Secondly, our approach makes use of source code and partial syntax information, without extracting the structure of the syntax tree and bytecode information. Therefore, the future work is needed to explore more code features and vulnerability characteristics.

Our dataset is not large enough to support more complex neural network structures. Training a complex structured neural network on an under-size dataset may lead to overfitting [36]. However, our approach has generality and can be easily applied to the larger datasets. Also, on the larger datasets, more code features and more complex model structures can be used to improve the performance of automatic program repair in smart contracts. This is a future work to be explored.

VIII. CONCLUSION

We present a dataset, containing over 10,000 method pairs on program repair for smart contracts, hopefully to advance the development of data-driven approaches in smart contract software engineering. By the learning relevant code changes in software history, we are the first to propose a statistical learning based approach called SMARTREP to provide one-line patches for program repair of smart contracts efficiently. We evaluate SMARTREP on the new dataset. SMARTREP is able to structurally repair 467/2,248 (42.9%) and completely repair 535/2,248 (23.8%) real bugs in testing set. It outperform the closest related work on program repair and the classical sequence-to-sequence models. Compared to existing automatic program repair approaches for smart contracts, SMARTREP is able to fix more types of vulnerabilities. It can also provide multiple security code recommendations supporting quite a few kinds of code editing operations for developers.

ACKNOWLEDGMENT

The work described in this paper was supported by the Key-Area Research and Development Program of Guangdong Province (2020B010164002) and the National Natural Science Foundation of China (61976061, 61902441, 61902105).

REFERENCES

- [1] Xiangfu Zhao, Zhongyu Chen, Xin Chen, Yanxia Wang, and Changbing Tang. The dao attack paradoxes in propositional logic. In *2017 4th International Conference on Systems and Informatics (ICSAI)*, pages 1743–1746. IEEE, 2017.
- [2] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25. IEEE, 2018.
- [3] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 531–548, 2019.
- [4] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68. IEEE, 2018.
- [5] Johannes Krupp and Christian Rossow. {teEther}: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, 2018.
- [6] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
- [7] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software “quantify the time and cost saved using reversible debuggers”. 2013.
- [8] Martin Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018.
- [9] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. Smart contract repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–32, 2020.
- [10] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 23–34. IEEE, 2020.
- [11] Yilin Wang, Xiangping Chen, Yuan Huang, Hao-Nan Zhu, and Jing Bian. An empirical study on real bug fixes in smart contracts projects. *arXiv preprint arXiv:2210.11990*, 2022.
- [12] Fabian Schär. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review*, 2021.
- [13] Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. Understanding code reuse in smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 470–479. IEEE, 2021.
- [14] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541, 2020.
- [15] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering*, 2020.
- [16] Y Ni, Chao Zhang, and T Yin. A survey of smart contract vulnerability research. *Journal of Cyber Security*, 5(3):78–99, 2020.
- [17] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.
- [18] Peng Qian, Zhenguang Liu, Qinming He, Butian Huang, Duanzheng Tian, and Xun Wang. Smart contract vulnerability detection technique: A survey. *arXiv preprint arXiv:2209.05872*, 2022.
- [19] Smart contract weakness classification and test cases. <https://swcregistry.io/>.
- [20] A community-developed list of software and hardware weakness types. <https://cwe.mitre.org/>.
- [21] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.
- [22] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.
- [23] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE, 2019.
- [24] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(09):1943–1959, 2021.
- [25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- [26] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, 2009.
- [27] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–30, 2019.
- [28] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 802–811. IEEE, 2013.
- [29] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 727–738, 2016.
- [30] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.
- [31] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 129–139, 2018.
- [32] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269. IEEE, 2018.
- [33] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.
- [34] Yuan Huang, Queping Kong, Nan Jia, Xiangping Chen, and Zibin Zheng. Recommending differentiated code to support smart contract update. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 260–270. IEEE, 2019.
- [35] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1695–1712. IEEE, 2020.
- [36] Xue Ying. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, volume 1168, page 022022. IOP Publishing, 2019.