Received: 29 November 2023 | Revised: 17 May 2024 | Accepted: 27 June 2024

DOI: 10.1002/smr.2712

RESEARCH ARTICLE - METHODOLOGY

Journal of Software: Evolution and Process WILEY

SGDL: Smart contract vulnerability generation via deep learning

Hanting Chu[1] | Pengcheng Zhang[1] | Hai Dong[2] | Yan Xiao[3] | Shunhui Ji[1]

[1]the College of Computer Science and Software Engineering, Hohai University, Nanjing, China

[2]Computing Technologies, RMIT University, Melbourne, Australia

[3]Cyber Science and Technology, Shenzhen Campus of Sun Yat-sen University, Shenzhen, China

Correspondence
Pengcheng Zhang, the College of Computer and Information, Hohai University, Nanjing, China.
Email: pchzhang@hhu.edu.cn.

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 62272145, U21B2016

Abstract

The growing popularity of smart contracts in various areas, such as digital payments and the Internet of Things, has led to an increase in smart contract security challenges. Researchers have responded by developing vulnerability detection tools. However, the effectiveness of these tools is limited due to the lack of authentic smart contract vulnerability datasets to comprehensively assess their capacity for diverse vulnerabilities. This paper proposes a Deep Learning-based Smart contract vulnerability Generation approach (SGDL) to overcome this challenge. SGDL utilizes static analysis techniques to extract both syntactic and semantic information from the contracts. It then uses a classification technique to match injected vulnerabilities with contracts. A generative adversarial network is employed to generate smart contract vulnerability fragments, creating a diverse and authentic pool of fragments. The vulnerability fragments are then injected into the smart contracts using an abstract syntax tree to ensure their syntactic correctness. Our experimental results demonstrate that our method is more effective than existing vulnerability injection methods in evaluating the contract vulnerability detection capacity of existing detection tools. Overall, SGDL provides a comprehensive and innovative solution to address the critical issue of authentic and diverse smart contract vulnerability datasets.

KEYWORDS
deep learning, generative adversarial network, smart contract, vulnerability injection

1 | INTRODUCTION

Since the inception of Ethereum, there has been a surge in blockchain security incidents, primarily caused by smart contract vulnerabilities. These incidents have not only resulted in significant economic losses but have also eroded users' trust in Ethereum.[1,2] For instance, the BEC smart contract was found to have an integer overflow vulnerability which was exploited by hackers.[3,4] They leveraged the flaw by inputting specific parameters to transfer large sums of cryptocurrency to their own address repeatedly in a brief period, which caused the BEC coin's market value to plummet.[5] In a similar vein, the DAO crowdfunding project lost a staggering 12 million ETH due to a reentrancy vulnerability.[6,7] This flaw ultimately led to the Ethereum hard fork event, which significantly shook the public's confidence in the "decentralization" value of blockchains.[8]

Researchers have developed a range of tools and methods to uncover security vulnerabilities in smart contracts.[9,10] Despite this progress, existing vulnerability detection tools have limitations as they are typically assessed using manually collected datasets with limited vulnerability samples, which cannot fully evaluate the effectiveness of the tools. To overcome this issue, scholars have begun proposing techniques to automate the creation of comprehensive vulnerability datasets. For instance, SolidiFi,[11] a vulnerability injection tool for smart contracts, introduces

Abbreviations: SGDL, smart generation deep learning; AST, abstract syntax tree; TFEI, target fragment extractor of integer; TFEE, target fragment extractor of exception; TFES, target fragment extractor of send; TFETX, target fragment extractor of tx.origin; TFET, target fragment extractor of timestamp; TFER, target fragment extractor of reentrancy..

targeted security vulnerabilities by injecting pre-defined vulnerability fragments into all potential locations of a smart contract. However, this approach has drawbacks as the injection process is not refined enough to consider the correlation between the vulnerability fragments and the injected contract. From our survey of past papers, we have noticed that research on smart contract vulnerability generation has been relatively scarce in the past three years. This phenomenon has prompted our reflection, and we believe the main reasons behind it include: 1. Technical challenges and emerging fields: The novelty of smart contracts and blockchain technology presents unique technical challenges for vulnerability generation. A deep understanding of smart contract programming languages such as Solidity, and how to efficiently generate vulnerabilities that cover a wide range of attack scenarios, are the primary difficulties in current research. 2. Increasing importance of security: As smart contracts are more widely adopted, their security issues have been given greater emphasis. Vulnerability generation, as a means to enhance the security of smart contracts, is gradually being recognized for its research potential and practical value. Additionally, the vulnerability fragments used lack diversity and authenticity. In summary, generating vulnerabilities in the emerging smart contract realm poses significant challenges for existing techniques.

1. *The uniqueness of the smart contract programming language*. Although some results have been achieved with vulnerability injection work against traditional programming languages, the differences between the languages prevent its feasibility in the smart contract space. Smart contracts are written in Solidity. It significantly differs from traditional programming languages, including require statements, event statements, and inter-calls with other contracts. These differences limit the use of traditional vulnerability generation techniques in the smart contract space.
2. *The existing inspection methods do not consider the correlation between injected vulnerabilities and contracts*. The current smart contract vulnerability generation method, SolidiFi,[11] employs an indiscriminate injection method. This process does not take into account the relationship between the vulnerability fragments and the original segments of the contracts targeted for injection. Specifically, it fails to evaluate whether the vulnerability fragment being injected is appropriate for the specific smart contract at hand. Instead of tailoring the approach to the individual contract, SolidiFi draws from the same pool of vulnerability fragments, randomly injecting various types into the original contract. Consequently, the smart contracts produced through this method lack both authenticity and realism.
3. *The existing datasets lack diversity*. In the domain of smart contract vulnerability detection, the supply of labeled datasets is notably scarce. Compounding this issue, the vulnerability fragments produced by current generation techniques are often strikingly similar, even though there may be a vast array of fragment pools. This uniformity results in considerable, yet meaningless duplication, with variations often being confined to slight differences in variable names. This lack of diversity in the vulnerable code that is generated not only diminishes the uniqueness of individual fragments but also impedes the precise evaluation of detection tool performance. As a consequence, developers find it difficult to gauge the effectiveness of these tools in an unbiased and accurate manner, presenting a significant challenge in the field.

To address the above problem, we propose SGDL, a novel method for generating authentic and diverse smart contract vulnerabilities using generative adversarial networks' powerful data-fitting capabilities. Specifically, SGDL addresses the problem of *overlooked correlation between injected vulnerabilities and contracts* through a vulnerability injection type determination algorithm. Moreover, to mitigate the issue of inauthentic and non-diverse vulnerability fragments generated by existing work, SGDL proposes a smart contract vulnerability fragment generation approach based on generative adversarial networks. This method enables the creation of a pool of authentic and diverse vulnerability fragments for subsequent injections. Finally, SGDL guides the injection of vulnerability fragments using an abstract syntax tree, ensuring the syntactic correctness of the vulnerability fragments within smart contracts.

Contributions. The main contributions of this paper are:

1. SGDL is the first study to apply deep learning techniques to smart contract vulnerability injection. To the best of our knowledge, this is the first such technique in the field. Automatic generation of vulnerability fragments using deep learning models significantly saves the labor cost required for manual construction and improves the diversity of fragments.
2. SGDL can inject seven types of high-risk vulnerabilities in smart contracts. It is the first injection tool supporting such a broad range of high-risk vulnerabilities, to the best of our knowledge. In addition, SGDL is the first tool that can inject *short address attack* vulnerability. Short address attack is a high-risk vulnerability existing in interactions between clients and the Ethereum blockchain. Huge numbers of Ethereum coins can be directly impacted by this vulnerability.*
3. We have evaluated the false negatives and false positives of 10 static analysis tools based on Ethereum smart contracts generated by SGDL.† The experimental results show that existing analysis tools can only detect some of the vulnerabilities generated by SGDL. This indicates that the use of SGDL can reveal more weaknesses of these analysis tools. In terms of diversity, it demonstrates an average improvement of 69.99% over existing methods, and in terms of authenticity, an average improvement of 62.2%.

2 | BACKGROUND AND MOTIVATION

2.1 | Solidity language

Solidity is currently the premier high-level programming language for developing Ethereum smart contracts. Its expressive power comes from being Turing complete, which enables it to represent any complex logic, a significant asset in the development process. In Solidity, functions are uniquely identified through function selectors, comprised of the first four bytes of the keccak-256 hash value of the function signature (including the function name, parameters, and return values). These selectors are essential for pinpointing the called function during smart contract invocation transactions. Solidity also supports multiple inheritance, providing robust design capabilities. Its official compiler, *solc*, creates a linear inheritance sequence, extending from the base contract to the most derived one. This ordered arrangement assists in the management of complex contract relationships, demonstrating Solidity's flexibility.

The language offers thorough error handling through the use of *require* and *assert* statements. If these statements are evaluated as false, an exception is thrown, and the transaction is rolled back. This mechanism helps maintain the integrity and reliability of the contract's logic. In terms of data storage, Solidity enables users to manually specify variable locations. There are three primary storage areas: (1) storage, for variables that require persistent storage. (2) Memory, where local variables are generally stored, releasing space after the transaction execution. (3) Calldata, designated for parameters and other call data, with space released after the function call. An essential feature of Solidity is the fallback function, employed when a contract is called without the specified function, or when the contract receives ether without a response function. The contract defaults to using the fallback function, ensuring that unexpected scenarios are handled gracefully.

In summary, Solidity's rich feature set, ranging from expressive logic representation and well-defined inheritance structures to precise error handling and flexible storage management, makes it the preferred choice for Ethereum smart contract development. Its design ensures both robust functionality and adaptability, supporting developers in creating secure and efficient contracts.

2.2 | Abstract syntax tree (AST)

Abstract syntax tree (AST) [12] is an essential intermediate representation of programs. AST is initially designed to simplify program compilation by shielding the programming language from cumbersome syntax rules and clearly showing the logical relationships within the code for application developers through a tree-like structure. This intermediate representation can well represent the characteristics of the source code. AST has two essential features. First, it does not rely on specific grammar.[13] No matter if the grammatical description of the sequential language is subsequently modified using the top-down syntax analysis technique (LL(1)) and bottom-up syntax analysis technique (LR(1)) or requires more complex modifications, the subsequent analysis steps will not be drastically altered as long as the source code can be successfully converted to AST. Second, it is language-independent and can handle a wide range of programming languages

AST supports not only the classic C/C++ and JAVA languages but also Solidity, the relatively new smart contract language studied in this paper.[14] On account of those features, AST is widely used as an important intermediate representation structure in many fields, such as compilers, and code obfuscation and compression.

3 | SMART CONTRACT VULNERABILITY TYPES AND RATIONALE FOR SELECTION

3.1 | Background on vulnerability types

Studies on the classification and standards of smart contract vulnerabilities have been extensive.[15] For instance, in 2017, Atzei et al[6] analyzed security vulnerabilities in Ethereum smart contracts, categorizing them into three levels: programming language, virtual machine, and blockchain. Similarly, the decentralized application security project (DASP) outlined 10 categories of high-risk smart contract vulnerabilities in 2018.[‡] More recently, Zhang et al[16] proposed the *JiuZhou* vulnerability classification framework, which extended the *IEEE Software Fault Tree classification*, detailing 49 types of vulnerabilities and their respective severity levels.

3.2 | Rationale for vulnerability selection

The selection of vulnerabilities was grounded in terms of the severity of vulnerabilities referenced from the DASP and *JiuZhou* frameworks. Both sources provide essential insights into the potential impacts of vulnerabilities on smart contracts, which aids developers in risk assessment. Furthermore, considering the frequency of vulnerabilities leading to smart contract security incidents (logged-in ETH DApp attacks[8]), and the availability of open-source vulnerability datasets, we narrowed our focus to seven key types of smart contract vulnerabilities.

The selected vulnerabilities are:

- Reentrancy vulnerability
- Timestamp dependency vulnerability
- Integer overflow and underflow vulnerability
- Unhandled exception vulnerability
- Unchecked external send vulnerability
- Short address attack vulnerability
- Use tx.origin for authentication vulnerability

Each vulnerability's technical details, illustrative examples, and justifications for inclusion based on their significance and past occurrences are provided in the following subsections.

3.2.1 | Reentrancy

The vulnerability is triggered by an external user address recursively calling the same contract function.[17,18] The default smart contract in the Solidity language contains a fallback function with no function name and parameters, which is automatically triggered when a transfer is received.[19] When a smart contract triggers a transfer operation, a reentrancy attack may occur before the function modifies the contract state variables. Therefore, the attacker takes advantage of the developer's negligence to make the program repeatedly execute the malicious code designed by the attacker in one transaction until the gas is exhausted, causing huge economic losses. As shown in Figure 1, a withdraw function in *contract Victim* is to perform the fallback function. The attacker initiates a transaction by attacking the contract by repeatedly calling the withdraw function in contract Victim until the victim's account balance is 0 or the gas is exhausted.

3.2.2 | Timestamp dependency

As the name suggests, the vulnerability stems from the global variable of the smart contract, the timestamp.[20] The timestamp of the block to which the smart contract belongs is available to developers as a global variable, which is determined by the mining system and allows a deviation

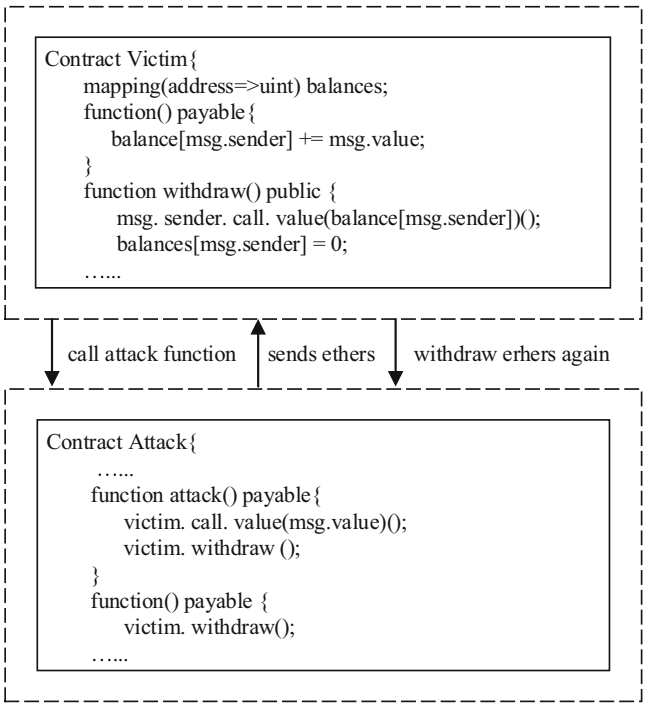


FIGURE 1 Reentrancy.

of 900 seconds so that miners can control the timestamp to some extent.[21] If the function implemented by the smart contract varies with the timestamp change, malicious miners can influence the result of the smart contract to a certain extent to meet their own needs. As shown in Figure 2, since there is *block.timestamp* in the *MyContract* contract as a condition to perform the key operation when the conditional statement *while* satisfies *release*! = *0 && block.timestamp > release*, the function will perform the important operation of the amount change. As a result, an attacker can attack the contract by manipulating the timestamp in the block to disguise the transaction, thus causing the theft of Ether.

3.2.3 | Integer overflow and underflow

Integer overflow occurs when there is a calculation operation against an integer variable in a statement or expression and the developer ignores the boundary value of the variable.[22] In such a situation, the variable value may exceed the upper or lower bound of the variable type, resulting in the variable value differing from what the developer expects and possible financial loss.[23] As shown in Figure 3, the variable *amount* on the fourth line is the multiplication of two *uint256* values. There is no overflow judgment on *amount*. If an attacker makes *amount* overflowed, then the attacker can bypass the code on the sixth line used to check the account balance. Through this vulnerability, the attacker can transfer a large number of tokens at a relatively low cost.

3.2.4 | Unhandled exception

This vulnerability stems from mutual calls between Ethereum smart contracts, e.g., using *<address>.send* or *<address>.call.value* statements to send tokens, or a call statement to call other contracts.[16] If there are exceptions, such as gas exhaustion, during such calls, these calls will be terminated, the state will be rolled back, and the false information will be returned to the calling contract.[24] If the caller uses a relatively low-level call statement (i.e., call and delegate call) and does not check the return value, the subsequent operations will continue, resulting in an implementation result different from what the developer expects.

3.2.5 | Unchecked external call

This vulnerability is related to external calls of smart contract.[25] If a contract is not carefully verified, when an external transfer request is initiated, unauthorized users can transfer tokens into the account, causing severe economic losses.

```
contract MyContract {
    function MyContract() public returns (uint tokens) {
        uint release;
        uint balance;
        while (release != 0 && block.timestamp > release) {
            tokens += balance;
            msg.sender.call.value(tokens);
        }
        return tokens;
    }
}
```

FIGURE 2 Timestamp dependency.

```
function batchTransfer(address[] _receivers, uint256 _value) public
whenNotPaused returns(bool) {
  uint cnt = _receivers.length;
  uint256 amount = uint256(cnt) * _value;
  require(cnt > 0 && cnt <= 20);
  require(_value > 0 && balances[msg.sender] >= amount);
  balances[msg.sender] = balances[msg.sender].sub(amount);
  …
  return true;
}
```

FIGURE 3 Integer overflow and underflow.

3.2.6 | Short address attack

A specific vulnerability emerges within Ethereum, originating from the auto-completion operation conducted by the Ethereum Virtual Machine (EVM). In the execution of smart contracts, which depend on the EVM,[26] the required input parameters for functions manifest within the virtual machine as fixed-length bytecodes.[15] This methodology, although standard, has its pitfalls. When function input parameters include address-based parameters lacking sufficient bits, they can be prone to short-address attacks.

The vulnerability is illustrated in Figure 4. First, a user deploys a smart contract containing the *sendCoin* function on the Ethereum blockchain. They then purchase 100 tokens through this contract and register an Ethereum account with the last two digits as 0 (e.g., 012…67800). An attacker, using a specific account (for example, _to: 012…678), calls the *sendCoin* function. As long as the transaction parameter _amount is less than the account balance of 100, the EVM, while packaging the transaction data, appends the first two digits (0) of the _amount parameter to the end of the _to parameter. In order to fit the length of the _amount parameter, the EVM then adds two more zeros at the end. This ultimately quadruples the _amount parameter when the contract is executed, revealing the vulnerability.

3.2.7 | Use *tx.Origin* for authentication

tx. origin is a global variable of Solidity, which is used to store the initial initiator of the overall transaction.[27] Incorrect use of *tx. origin* would invalidate identity verification.[28] It is not generally recommended to use *tx. origin* for identity verification, as it can lead to phishing attacks. The attack initiator will induce the victim to initiate a transaction to a malicious contract that has been extracted and deployed. Then the malicious contract will call the method in the contract deployed by the victim. At this time, the initial initiator of the transaction is the victim. That is, the address of the victim is stored in *tx. origin*, so malicious contracts can bypass the identity verification using *tx. origin* and perform some sabotage. As shown in Figure 5, an attacker can deploy an attack contract, including a fallback function that calls the *sendTo* function in *MyContract*. Since *tx. origin* == *owner* is used as the judgment condition in the *MyContract* contract, *tx. origin* represents the address from which the transaction was initially sent. By attacking the contract in the form of disguising the identity, the attacker can obtain the transfer qualification, thereby causing ether theft.

```
function sendCoin(address _to, uint256 _amount)  returns(bool) {
 if(balances[msg.sender]  <  _amount)
   return false;
 balances[msg.sender] = balances[msg.sender].sub(amount);
 balances[_to] = balances[_to].add(amount);
 Transfer(msg.sender, _to, _amount);
 return true;
}
```

FIGURE 4 Short address attack.

```
contract MyContract {
   address owner;
   function MyContract() public {
      owner = msg.sender;
   }
   function sendTo(address receiver, uint amount) public {
      require(tx.origin == owner);
      receiver.transfer(amount);
   }
}
```

FIGURE 5 Use *tx.origin* for authentication.

4 | DESIGN OF SGDL

4.1 | Overview of SGDL

SGDL is a deep learning-based Ethereum smart contract vulnerability injection tool based on a supervised learning paradigm. It can inject seven types of critical vulnerabilities. Figure 6 shows the overall workflow of SGDL, which includes four steps: data collection, vulnerability type judgment, vulnerability fragment generation, and vulnerability injection. First, we collected the vulnerability fragment dataset for model training. Specifically, *SGDL* employs a set of rule-driven snippet extractors to extract corresponding vulnerable code snippets from smart contract source code and constructs a dataset of vulnerable code snippets. For different vulnerability types, SGDL realizes injection vulnerability type judgment through feature sequence extraction, contract information standardization, and vulnerability type classification. Feature sequence extraction is to convert the AST generated from the source code of a smart contract into sequence information. Contract information standardization is to perform normalization on the feature information. Vulnerability type classification is to use a classification model to determine the type of vulnerability type to be injected. Next, SGDL pre-trains a vulnerability fragment generation model by extracting 80% of the training dataset so that the model can learn the grammatical information of Solidity. It then utilizes various types of vulnerability fragments to fine-tune the model for the respective vulnerability type. A model that can effectively generate various vulnerability fragments is therefore obtained. Finally, an AST-based analysis method is employed to determine the vulnerability injection locations and then sequentially inject the vulnerability fragments into the appropriate place.

4.2 | Data collection

The source of the dataset is twofold: First, we used the keywords of *smart contract vulnerability*, *vulnerable smart contracts*, *buggy smart contracts*, and *smart contracts defects* to search smart contracts on GitHub and Gitter chat room[¶]; Second, we used *Karl*[#] to collect new smart contracts from the Ethereum blockchain website,[**] where *Karl* is a tool used together with *Mythril* for real-time blockchain monitoring. They can provide addresses of smart contracts that may cause vulnerabilities. For data labeling, we utilize *Mythril*, *Slither*, *Oyente*, and *SmartCheck* to detect vulnerabilities in the collected contracts. When three out of four tools report issues with a specific line of the contract, we manually verify and label the specific type of vulnerability after validation. Subsequently, we add the labeled contracts to the vulnerability dataset.

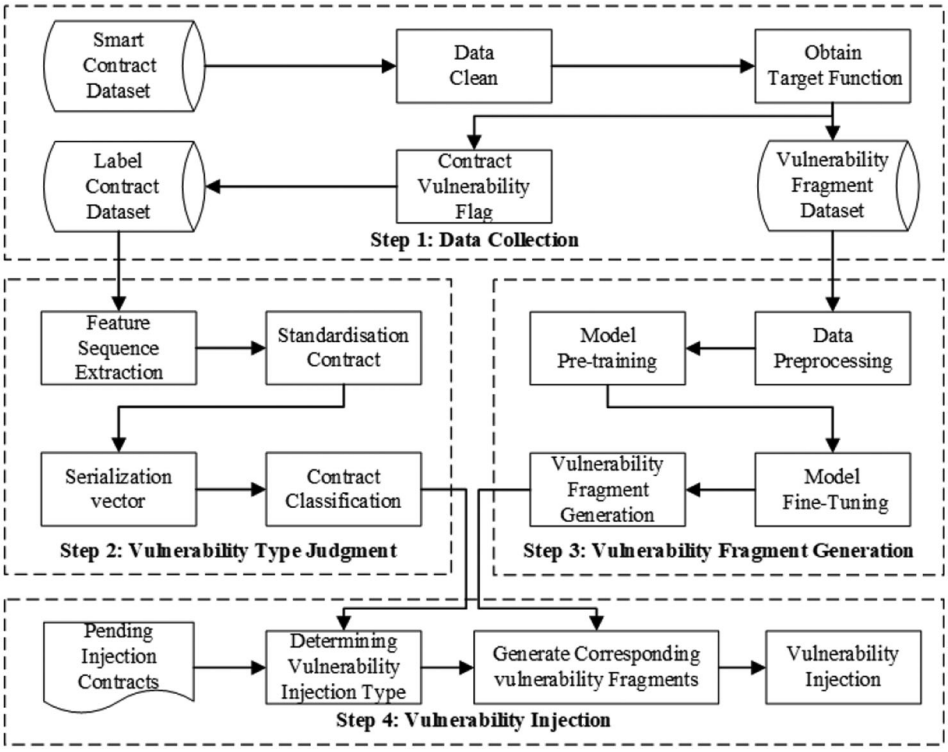


FIGURE 6 Overview of SGDL

Collecting sufficient samples is the prerequisite for the model to learn the inherent laws of authentic smart contract vulnerabilities. In real smart contract vulnerability fragment samples, in addition to including the statement that causes the vulnerability, it is also necessary to ensure that the rest of the content is associated with the statement that causes the vulnerability. The sample data for model training should contain as comprehensive vulnerability information as possible. In this regard, we defined a series of extractors to automatically collect comprehensive code fragments for each vulnerability type from smart contract source code, the details of which are depicted below.

4.2.1 | Target fragment extractor of integer overflow and underflow (TFEI)

TFEI first retrieves statements or expressions related to integer operations in the smart contract as target objects. If the retrieved object is in a safe form, such as SafeMath.add (var0,var1), *TFEI* extracts the parameters and converts them into general integer operations using operators. Then, *TFEI* analyzes the state of the target object. If integer operations appear in available assignments or variable definitions, *TFEI* extracts the variables assigned in the statement and extracts variable statements containing the same method together. If an integer operation appears in the evaluation condition of a require statement, *TFEI* skips the require statement and extracts the subsequent operation statements. If an integer operation appears in the evaluation condition of an if statement, *TFEI* extracts the conditional statement and the subsequent operations together. If an integer operation appears in a loop body, *TFEI* extracts the loop condition together.

4.2.2 | Target fragment extractor of unhandled exception (TFEE)

TFEE first retrieves the calling statements of the lower level as the target object. If the call statement appears in the conditional judgment of a require statement or an if statement, *TFEE* skips the judgment condition when extracting the code fragment, only retaining the call statement and its subsequent relevant operations. If a Boolean variable is defined in the contract to store the return value, *TFEE* will extract the statements that perform judgment operations on the Boolean variable and their subsequent relevant operations together.

4.2.3 | Target fragment extractor of unchecked external send (TFES)

TFES first uses externally initiated transfer statements as the target object for retrieval. If there is identity verification before the target statement, *TFES* skips the verification statement when extracting the code fragment. If there is an unconditional judgment statement before the transfer statement without identity verification functionality, *TFES* directly extracts these statements.

4.2.4 | Target fragment extractor of use *tx.origin* for authentication (TFETX)

TFETX first retrieves statements that use tx.origin for identity verification as the target statements, and then analyzes the components of the target statements. If tx.origin==msg.sender is used for identity verification, *TFETX* replaces the verification content with a safer form. Finally, *TFETX* extracts the identity verification statements and their subsequent relevant operational statements together from the function.

4.2.5 | Target fragment extractor of timestamp dependency (TFET)

TFET first takes block.timestamp and now as the target objects. If the target objects appear in conditional statements, *TFET* extracts the conditional statements and their related operations. If the target objects involve the return value of a function, *TFET* will extract the relevant statements together.

4.2.6 | Target fragment extractor of reentrancy (TFER)

TFER first searches for transfer statements as the target statements, and then checks whether there are statements that adjust balance variables in the transfer statements. If both of these statements exist in the function body, *TFER* extracts the corresponding code fragment as a vulnerability fragment and ensures that the balance decreasing statement appears after the transfer statement.

4.2.7 | Target fragment extractor of short address attack (TFEA)

TFEA first searches for transfer statements in the smart contract. If the variables representing the address and the number of tokens in the statement can be controlled by user inputs, *TFEA* takes the transfer statement and the function definition as the target objects. Then, *TFEA* analyzes the conditional statements before the target statement. If there are validation operations on the length of the address variable, *TFEA* omits the validation operation and finally extracts the corresponding vulnerability fragment.

4.3 | Vulnerability type judgment

To obtain a vulnerability fragment appropriate for injecting into an original contract, we first need to analyze the syntax and semantic information of the original contract to determine the type of vulnerability suitable for injection. Since there are many similarities in the code structure and implemented functions of smart contracts with the same vulnerabilities, SGDL adopts the idea of classification to achieve the function of determining the type of vulnerability injection.

4.3.1 | Feature sequence extraction

The ideal feature sequence needs to contain rich structural information of a smart contract. Since the syntax restrictions of the Solidity language are not reflected explicitly in the smart contract source code, simply analyzing the source code cannot obtain sufficient information. In this regard, an abstract syntax tree showing the intermediate structure of the code can solve this problem.

We adopt ANTLR[††] (AN-other Tool for Language Recognition) to parse those function fragments into AST. ANTLR is a powerful parser generator for translating structured text and analyzing language. Here, we deploy a depth-first search to serialize the AST of the function fragments.

Figure 7 shows a sample abstract syntax tree generated by the ANTLR compilation. From the converted results shown in Table 1, the comparison shows that the two statements, which differ in source code form by only a few characters, show a large difference when transformed into a sequence of features.

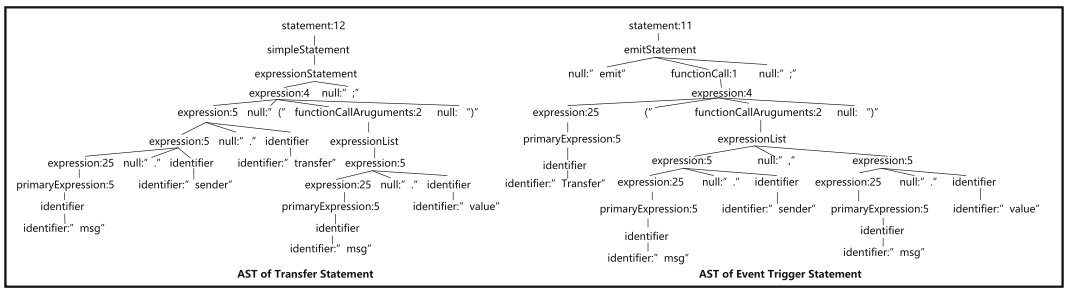


FIGURE 7 AST comparison chart.

TABLE 1 Sequence of features corresponding to transfer statements and event statements.

Source code	Msg.Sender.Transfer (msg.Value)	Emit transfer (msg.Sender,msg.Value)
Feature Sequence	{statement transfer simpleStatement "(" expressionStatement functionCallArgument expression expressionList expression expression expression expression expression primaryExpression primaryExpression msg msg "." "." value sender ")" "." ";"}	{statement primaryExpression emitStetement msg emit "." functionCall sender expression "," expression expression primaryExpression expression transfer primaryExpression "("primaryExpression functionCallArgument "." expressionList value expression ")" expression ";"}

4.3.2 | Contract information standardization

The naming of variables in smart contracts is specified by the developer as in traditional programming languages. To reduce the impact of low-frequency words on the model, contract normalization is required to remove some task-irrelevant contract information. This is performed as follows: 1) for constants in the contract, character constants are replaced with StringLiteral and numeric constants with DecimalNumber; 2) for local variables with no real meaning, the word "simpleVar" is used as a uniform replacement.

4.3.3 | Feature vector generation and classification

After the above steps, the smart contract has been transformed into a sequence of features. To fit the input of the machine learning classification algorithm, it is also necessary to map each word element in the feature sequence into a high-dimensional word vector. SGDL chooses the Fasttext model[29] to implement the generation of the word vector. Compared with Word2vec, Fasttext uses n-grams to represent each word. This processing method enables the model to process words that do not exist in the corpus, which can better process variable information of smart contracts. SVM performs classification by finding decision boundaries that best separate different data classes. SVM is highly flexible and accurate, can avoid overfitting, and maintains promising performance when faced with high-dimensional data features. We choose the SVM classifier for determining vulnerability injection type operations.

4.4 | Vulnerability fragment generation

After determining what type of vulnerability is suitable for injection, generating the corresponding type of vulnerability fragment for subsequent injection operations is required. A limited pool of vulnerability fragments will make the generated smart contract vulnerabilities more detectable and less authentic. In this regard, a deep learning model can learn the inherent patterns of original vulnerability fragments to generate more authentic and diverse smart contract vulnerability fragments.

4.4.1 | Data preprocessing

Before feeding the smart contract vulnerability fragment data into the deep learning model for training and testing, we need to preprocess the vulnerability fragments to remove the effects of some variables.

The first step is to reduce the length of the sequence by traversing the leaf nodes of the AST and then by introducing the concept of code idioms, which refers to groups of words in the source code that do not need to be segmented and generally have program-specific semantics.[30] The code conventions selected in this paper are divided into two categories, one is "tx.origin", "msg.sender", "adddress(0)", and so on. The other category is "balance[to]", "balance[from]", and other phrases that appear frequently and carry certain functional information. For low-frequency words that are not part of the code idioms and Solidity keywords, standard variable names such as VAR $\{n\}$, Fun $\{n\}$, etc., where n represents the order in which variables appear in the current segments.

4.4.2 | Model training

To address the challenge of generating structure-rich vulnerability information, we employ LeakGAN, which has been proven to be proficient in handling long text generation tasks .[31] Our primary contribution lies in the novel adaptation and fine-tuning of LeakGAN's architecture to specifically cater to the problem of smart contract vulnerability generation.

We have made advancements to the conventional LeakGAN framework by tailoring the discriminator and generator components to effectively capture the intricate vulnerability patterns present in smart contracts. Our modified model incorporates a meticulously trained discriminator that exhibits the ability to discern subtle features associated with vulnerabilities. The motivation behind using the Sigmoid function in our enhanced feature extractor is to accurately capture the subtle distinctions between different types of vulnerabilities in smart contracts, which often exhibit complex and nuanced code patterns. By mapping the output values to the (0,1) interval, the Sigmoid function allows for a more precise and granular analysis of these patterns, facilitating identifying and classifying vulnerabilities with greater specificity. The rationale for this design is rooted in the need for a robust classification system that can effectively handle the intricate and specific features of smart contract code. Traditional activation functions often fail to provide a solution for detecting subtle changes, which can lead to misclassification or supervision of potential vulnerabilities. Integrating the Sigmoid function enhances the model's ability to differentiate closely but crucially changing data, thereby

achieving more accurate vulnerability classification. This approach has improved the performance of our vulnerability model. This aspect is of utmost importance, given the intricate and complex nature of smart contract security.

For the discriminator, we have introduced an enhanced feature extractor that is more attuned to the nuances of vulnerability-laden sentences s, refining the output features $f = F(s)$ for superior classification using the sigmoid function as shown in Equation 1. The generator, equipped with a specialized Manager module, now processes feedback from the discriminator more effectively. This refined process allows for a more precise generation of the embedding vector of the target, leading to more accurate and realistic vulnerability segments as depicted in Equations 2 to 7. A pivotal aspect of our contribution is the introduction of domain-specific pre-training, followed by meticulous fine-tuning procedures that allow the model to generate authentic vulnerability fragments. This process has been tailored for various types of smart contract vulnerabilities, significantly enhancing the relevance and accuracy of the generated text. Furthermore, the interplay between the discriminator and generator has been fine-tuned to provide a more nuanced reward system, as shown in Equation 7, which better aligns with the specific goal of vulnerability generation and description in smart contracts.

By focusing on these enhancements, we have extended the capabilities of LeakGAN beyond its original scope, providing a tool that is not only effective in generating long texts but is also intricately tuned for the high-stakes field of smart contract vulnerabilities. The overall network structure, while built on the foundations provided by LeakGAN, has been significantly adapted to suit our needs. Figure 8 illustrates the modified architecture, which, through our contributions, offers an innovative approach to vulnerability fragment generation in smart contracts. Our unique contribution, therefore, is not the use of LeakGAN perse, but rather the significant enhancements and domain-specific optimizations that have been introduced, ensuring that the model is well-suited for the targeted task of vulnerability generation in smart contracts.

$$D(s) = sigmoid(f) \tag{1}$$

Where $D(s)$ represents the output of the discriminator, tasked with distinguishing between real and generated data. f represents the output features from the enhanced feature extractor within the discriminator. $sigmoid(f)$ is the Sigmoid function used to map the input to the (0,1) interval, thereby enhancing the output of the feature extractor.

$$g_t, h_t^M = LSTM\left(f_t, h_t^M, \theta_M\right) \tag{2}$$

Where g_t is the output of the generator at time step t. f_t represents the input at time step t. h_t^M represents the hidden states of the Manager modules in the generator at time step t. θ_M represents the set of parameters used in the Manager module.

$$g_t = \frac{\widehat{g}_t}{\|\widehat{g}_t\|} \tag{3}$$

Where $\widehat{g}_t$ is the unnormalized target embedding vector. $\|\widehat{g}_t\|$ is the norm of the target embedding vector.

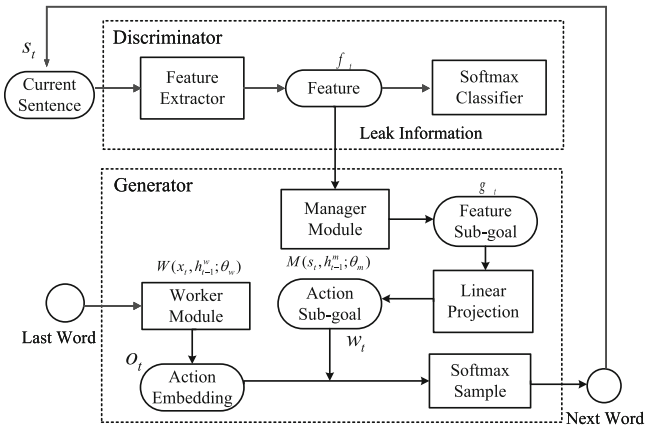


FIGURE 8 LeakGAN network structure.

$$w_t = W_\psi \left(\sum_{i=1}^{c} g_{t-i} \right) \tag{4}$$

Where w_t is the word embedding vector output by the Worker module. W_ψ is the linear projection layer used to map the target embedding vector to the word space. $\sum_{i=1}^{c} g_{t-i}$ represents the summation of generator outputs from time step $t-i$ to time step $t-1$.

$$Q_t, h_t^W = LSTM\left(x_t, h_{t-1}^W, \theta_W\right) \tag{5}$$

Where Q_t, h_t^W represent the output and state of the Worker module in the generator, respectively. x_t is the input at the current time step. θ^W is the parameters of the Worker module. h_{t-1}^W represents the hidden state of the Worker module in the generator at time step $t-1$.

$$G(s_{t+1}|s_t) = sigmoid\left(\frac{Q_t w_t}{\alpha}\right) \tag{6}$$

Where $G(st+1|st)$ represents the probability of generating the next state s_{t+1} given the current state s_t. α is a tuning parameter.

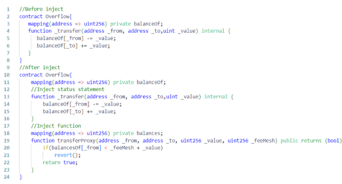
$$r_t^W = \frac{1}{c} \sum_{i=1}^{c} d_{\cos}(s_t - s_{t-i}, g_{t-i}(\theta_M)) \tag{7}$$

Where r_t^W represents the similarity score between the generated word and the target embedding vector at the current time step. $d_{\cos}$ denotes the cosine distance function.

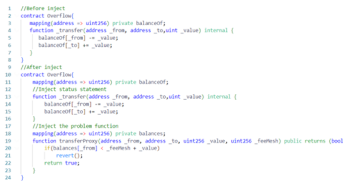
4.5 | Vulnerability injection

SGDL first counts the number of vulnerability fragments and their fragment types that need to be injected into an original contract then iterates through the AST of the original contract. When a stateVariableDeclaration or functionDefinition node is accessed, the offset value of the corresponding starting symbol in the contract source code is recorded. The offset value will be used as the marker for subsequent vulnerability injection. The source code level vulnerability injection is achieved through text interception and splicing to generate a smart contract with vulnerabilities. However, a common failure mode occurs when the generated variable names do not match those in the existing codebase. This mismatch can cause the smart contract to fail due to undefined variables and fail at runtime. Based on the aforementioned operations, vulnerability injection is implemented at the source code level, generating smart contracts with vulnerabilities. The results after injection are displayed in Figure 9. Finally, the type of vulnerability is marked above the injected method body in the form of a comment, facilitating analysis. Algorithm 1 shows the whole process of vulnerability injection.

Given the source code of an original smart contract *originalCode* and a pool of vulnerability fragments *snippetSet*, we aim to inject the smart contract with appropriate types of vulnerability fragments. First, we determine the types of vulnerability fragments to be injected, and then obtain the required function-level contract fragment and state variable of the determined vulnerability fragment (lines 1-4). We then convert the source code of the original smart contract into an AST (line 5). We iterate through the AST of the original contract (line 6). If the node of the state variable



(A) *A correct injection example.*



(B) *An incorrect injection example.*

FIGURE 9 Examples of vulnerability injection result.

and key function is accessed, we record the node (lines 7-14). If the nodes of state variables and key functions are accessed, the offset value of the corresponding node in the source code of the contract is recorded (lines 11-14). After obtaining the specific location for the vulnerability injection, the state variables and function fragments are injected into the corresponding offset location (lines 15-23). Finally, the smart contract injected with appropriate vulnerability fragment(s) is generated (line 24).

Algorithm 1: Smart contract vulnerability injection algorithm

Ensure: *originalCode*: Source code for smart contracts to be injected; *snippetSet*: The vulnerability fragment to be injected.

Require: *injectedCode*: Source code of the injected smart contract.

1: **for** *snippet* in *snippetSet* **do**
2: *stateVariableSet* ← *stateVariable*
3: *functionSet* ← *function*
4: **end for**
5: *ast* ← *code*
6: Initialize *stateVariableLocList*, *functionLocList*
7: **for** *node* in *ast* **do**
8: **if** *node* in *stateVariableDeclaration* **then**
9: **Push** *node* to *stateVariableLocList*
10: **end if**
11: **if** *node* in *functionDefinition* **then**
12: **Push** *node* to *functionLocList*
13: **end if**
14: **end for**
// Inject State Variable
15: **for** *stateVariable* in *stateVariableSet* **do**
16: *stateVariableLoc* ← *stateVariableLocList*;
17: **Inject** *stateVariable* into *originalCode* at *stateVariableLoc*
18: **Push** *stateVariableLoc*, *stateVariableLoc*, *type* in *injectLog*
19: **end for**
// Inject Vulnerable Function Fragments
20: **for** *function* in *functionSet* **do**
21: **Inject** *function* into *originalCode* at *functionLoc*
22: **Push** *functionLoc*, *function*, *type* in *injectLog*
23: **end for**
24: **return** *injectedCode*, *injectLog*

Evaluation

4.6 | Dataset

We obtained a total of 66,103 smart contract datasets by crawling the source code of real smart contracts deployed on Etherscan.io. In order to build the dataset for model training, the collected raw data were processed using the data filtering and collection means mentioned in Section 4 to obtain the smart contract dataset with vulnerability labels and the vulnerability fragment dataset. The details of this smart contract dataset are shown in Table 2. The details of the smart contract vulnerability fragment dataset are shown in Table 2. Data pre-processing and feature sequence extraction of smart contracts were realized in JAVA, while feature sequence vectorization and neural network-related operations were implemented in Python. For our final model configuration, we used 120 rounds of adversarial training, 50 steps of pre-training for the generator, 10 steps of pre-training for the discriminator, generated 6,400 samples per time, set the learning rate of the generator optimizer at 0.0001, the learning rate of the discriminator optimizer at 0.0001, the update rate of the rollout model at 0.8, and the batch size of 64. We randomly divided 80% of the dataset into a training set and the remaining 20% into a test set. We have released our code at: https://github.com/Tourneso/SGDL.

Our evaluation aims to respond to the following questions:

TABLE 2 Smart contracts dataset

Vulnerability type	Total contracts	Fragments number
Integer Overflow and Underflow	36,539	133,846
Unhandled Exception	6,769	44,388
Unchecked External Send	6,602	40,182
Use tx. origin for authentication	18,188	89,584
Timestamp Dependency	11,077	66,340
Reentrancy	5,663	4,128
Short Address Attack	2,732	3,196

- *RQ1*: How does the generative model's performance fare in generating vulnerability fragments?
- *RQ2*: Is the vulnerability injection type determination algorithm effective?
- *RQ3*: Are the smart contract vulnerability fragments generated by *SGDL* authentic and diverse?
- *RQ4*: Can the *SGDL* approach provide a more objective performance assessment for existing smart contract vulnerability detection tools?
- *RQ5*: Can the vulnerability fragments unable to be detected by a detection tool be activated?

4.7 | Evaluation metrics

Evaluation metrics for gauging the performance of text generation models are well-established across various domains.[32,33] In our analysis, we adopt two specific metrics: for synthetic data, we apply the negative log-likelihood, denoted as NLL_{gen}, along with its related term, labeled as NLL_{oracle} .[34] The equations to compute NLL_{gen} and NLL_{oracle} are detailed below:

$$NLL_{oracle} = -\mathbb{E}_{Y_\theta \sim P_\theta}\left[\log P_\gamma(y_1,...,y_T)\right] \tag{8}$$

$$NLL_{gen} = -\mathbb{E}_{Y_\gamma \sim P_\gamma}[\log P_\theta(r_1,...,r_T)] \tag{9}$$

θ represents the model's parameters, while Y_θ signifies text sequences generated using these parameters. P_θ embodies the model's probability distribution, and P_γ stands for the genuine data distribution. Meanwhile, y_T designates true text sequences.

Generally, *NLL_gen* measures sample diversity, while *NLL_oracle* is more sensitive to sample quality. For the real data set, we also employ *NLL_gen* to measure sample diversity. However, since *NLL_oracle* cannot evaluate the quality of real data, we use the Bilingual Evaluation Understudy (BLEU)[35] score to measure sample quality. BLEU is a commonly used metric for evaluating the quality of outputs generated by machines. Its computation is based on the degree of n-gram overlap between the machine-generated samples and reference samples. BLEU compares the machine-generated samples with multiple reference samples and then calculates the number of overlapping n-grams between the generated samples and the reference samples. A higher degree of overlap indicates higher quality in the generated samples.

4.8 | RQ1: generative Model's evaluation

To answer *RQ1*, we conducted comparative studies, substituting *LeakGAN* with several generation models: an LSTM trained with Maximum Likelihood Estimation (MLE), *SeqGAN*[36], and *RankGAN*.[37] MLE identifies model parameters that amplify the likelihood probability of the training data, enhancing the model's proficiency in both modeling input sequence data and boosting its generation or prediction capabilities. By maximizing the likelihood of the training data, the LSTM model is optimized to produce more accurate and representative sequences. *SeqGAN* is a generative adversarial network (GAN)-based model specifically designed for text generation tasks. It uses a reinforcement learning approach to train the generator network. *SeqGAN* treats the text generation problem as a reinforcement learning problem. The generator network is trained using policy gradient methods to maximize the expected reward, which is obtained from a discriminator network. *RankGAN* is also a GAN-based model for text generation. It focuses on generating high-quality text by using a ranking objective during training. The core idea of *RankGAN* is to generate text sequences that can rank higher than other randomly sampled sequences. The insights garnered from these experimental outcomes will shed light on *SGDL*'s efficacy compared to state-of-the-art approaches.

We compared the performance of the aforementioned models to determine the most suitable neural network model for smart contract vulnerability generation. Initially, we fine-tuned the hyperparameters of each model, such as learning rate, batch size, layer depth, and hidden units.

For consistent evaluation, we employed identical data parameters across various models, including training and testing sets, as well as the maximum sequence length, to assess the quality of the generated text. To ensure the fairness of the experiments, each method utilized the same data processing and feature extraction methods to generate the same types of vulnerabilities. This standardized approach allowed us to make fair and reliable comparisons between different models and hyperparameter settings.

Our experiments utilized synthetic data, comprising text lengths of 20 and 40 words. This synthetic dataset encompasses 20,000 samples, with each half generated from a distinct oracle-LSTM instance. An oracle-LSTM is an LSTM model renowned for generating superior text data.[36] Opting for shorter text lengths, like 20 words, expedites the evaluation process since briefer texts are quicker to generate. Moreover, these concise excerpts aptly represent the model's overarching performance, encapsulating routine phrasal and sentence structures. Conversely, a 40-word length ensures the inclusion of ample contextual information in the assessment. Such extended texts can aptly portray the model's prowess in managing extensive sentences or paragraphs, thereby shedding light on its capability to address long-term dependencies. By assessing two distinct text lengths, we harmonized the implications of short-term and long-term context, offering a holistic perspective on the model's efficacy. We employed the NLL_{oracle}[34] metric to gauge sample quality. Table 3 showcases the overall *NLL* performance, with *LeakGAN* surpassing its counterparts in both scenarios. Notably, *LeakGAN*'s superiority amplifies as the sequence length expands, signifying its elevated sample quality and overall proficiency in the realm of smart contract vulnerability generation.

Partitioning the data, 80% served as the training set, with the remaining 20% designated as the test set. Our evaluation metrics spanned BLEU-2 to BLEU-5 scores,[35] with the outcomes delineated in Table 4. *LeakGAN* showcased marked advancements over the benchmark models across all evaluated metrics. Stellar BLEU scores consistently emerged, underscoring the sentences generated by LeakGAN's prowess in mirroring the local characteristics of authentic texts.

Answer to RQ1: *SGDL* has proven to be effective for smart contract vulnerability generation, achieving the best performance with LeakGAN. The *NLL_gen* value of *SGDL* is 0.26 higher than the best performance of all baseline methods. This indicates that *SGDL* can generate samples of higher quality and demonstrate strong performance in capturing long-term dependencies.

4.9 | RQ2: validity experiments

To answer RQ2, we set up comparison experiments to verify the effectiveness of the SVM classification algorithm.[38] We select the distance-based KNN algorithm[39]and LightGBM[40] that is built upon an iteratively trained weak classifier, as the baseline algorithms. First, we construct a validation dataset containing labeled and manually fixed vulnerabilities, with different vulnerabilities generating different probabilities. The collected validation dataset will be filtered to prevent excessive sample imbalance. Table 5 shows the final data collection. Then, the cosine similarity between the smart contract after fixing the vulnerability and each benchmark sample fragment obtained in Section 4.2 is calculated. Subsequently, since the number of samples with the least number of vulnerability types in the validation set is 10, the 10 samples with the highest similarity are selected. The vulnerability labels with the highest occurrences in the selected samples are counted as the vulnerabilities suitable for injection. The match is successful if the calculated vulnerability type ideal for injection is the same as the labels of the experimental samples. The essence of the accuracy of the injected vulnerabilities to be verified in this experiment is to judge how well the classification model works, so in this experiment, we use *Precision*, *Recall*, and *F1 scores* as evaluation metrics. When the model is trained for a certain type of vulnerability, samples with other vulnerability labels will be considered as negative, thus transforming into a binary classification problem. The experimental results of *Accuracy*, *Recall*, and *F1 scores* are shown in Figure 10.

TABLE 3 The *NLL* scores on general text generation

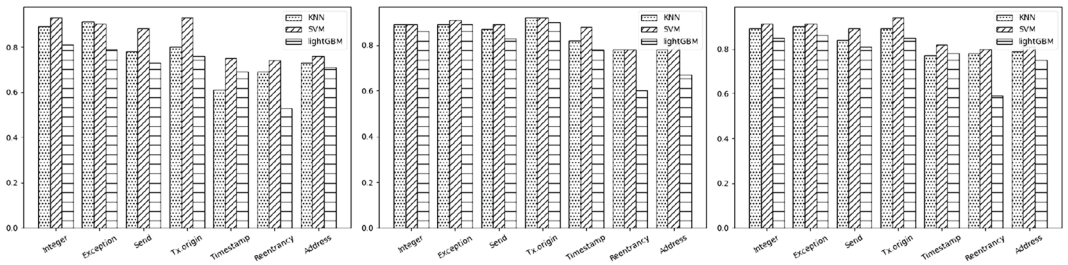
Length	MLE	SeqGAN	RankGAN	LeakGAN
20	9.038	8.736	8.247	7.038
40	10.411	10.310	9.958	7.191

TABLE 4 The BLUE scores performance

Method	MLE	SeqGAN	RankGAN	LeakGAN
BLEU-2	0.855	0.844	0.871	0.908
BLEU-3	0.529	0.509	0.643	0.746
BLEU-4	0.227	0.393	0.412	0.579
BLEU-5	0.132	0.202	0.291	0.384
NLL_gen	3.145	2.977	2.676	2.416

TABLE 5 Classification model validation dataset.

Vulnerability type	Number
Integer Overflow and Underflow	30
Unhandled Exception	30
Unchecked External Send	20
Use tx. origin for authentication	15
Timestamp Dependency	15
Reentrancy	15
Short Address Attack	10



(A) *Precision* (B) *Recall* (C) *F1_Score*

FIGURE 10 Evaluation indicator results.

From the experimental results, all three classification algorithms achieved promising results for the judgment of various vulnerabilities. This indicates that the feature extraction method proposed by SGDL can better retain the valuable syntactic and semantic information of smart contracts and accurately discover the types of vulnerabilities that are more suitable for injection into contracts. In addition, the SVM algorithm has the best performance among the three classification algorithms.

Answer to RQ2: *SGDL* enables effective determination of the seven vulnerability types and achieves optimal performance on SVM.

4.10 | RQ3: quality assessment

To answer RQ3, we evaluate the diversity and authenticity of the method in terms of the similarity of the generated vulnerability contracts and the quality of the generated contracts, respectively.

4.10.1 | Diversity evaluation

Suppose the pool of exploit fragments used for injection is not highly diverse. In this case, those samples of exploits used for repeated injections can easily generate many identical evaluation results. The more repetitions of vulnerability fragments that are not easily detected in the pool, the worse the overall assessment will be, and vice versa, making the performance assessment results of the inspection tool biased.

We calculate the similarity between fragment samples by comparing a selection of fragments in the fragment pool. If the similarity between a pair of fragments is greater than 95%, then one of the fragments is considered invalid and is removed from the collection. Once the comparison is complete, the remaining fragments in the vulnerability pool are considered valid, demonstrating the results of the different efforts in diversity comparison by comparing the number of reasonable fragments. The results of the experiments are shown in Table 6, with each result presented as <number of valid fragments/total fragments>.

From the experimental results, the *SGDL* method generates a higher percentage of valid fragments and has a more diverse representation of vulnerability fragments. *SolidiFi* open-source vulnerability fragment pool has many duplicate fragments that are identical except for the variable names, so the diversity results are less than *SGDL*.

TABLE 6 SGDL and SolidiFi diversity and authenticity results (* indicates that a tool does not cover this type of vulnerability)

	Diversity experiment		Authenticity experiment	
Vulnerability type	**SGDL**	**SolidiFi**	**SGDL**	**SolidiFi**
Integer overflow and underflow	86%	12.5%	82%	5%
Unhandled exception	82%	16.67%	80%	12.5%
Unchecked external send	86%	8.8%	70%	8.8%
Use tx. origin for authentication	78%	10%	72%	7.5%
Timestamp dependency	76%	7.5%	72%	7.5%
Reentrancy	82%	16.67%	58%	16.67%
Short address attack	86%	*	70%	*

```
function bug_intou3() public{
    uint8 vundflw =0;
    vundflw = vundflw -10;   // underflow bug
}
function bug_unchk_send1() payable public{
     msg.sender.transfer(1 ether);  //unchecked send
}
function unhandledsend_unchk2(address payable callee) public {
    callee.send(5 ether);                //unhandled exceptions
}
```

FIGURE 11 Example of a single component vulnerability fragment.

```
function bug_unchk7() public{
address payable addr_unchk7;
if (!addr_unchk7.send (10 ether) || 1==1)
    {revert();}
}
```

FIGURE 12 Example of an expression of unnatural vulnerability fragment.

4.10.2 | Authenticity evaluation

Uniquely diverse vulnerability fragments generated through diversity judgment must have a certain level of authenticity to be applied to vulnerability generation tasks. Real-world smart contract vulnerability fragments have two characteristics: (1) the presence of vulnerabilities that can be exploited by attackers in the fragment; (2) the vulnerability must appear natural, similar to real smart contract vulnerabilities deployed on the Ethereum network. We use both manual verification and quantitative metrics to evaluate the authenticity. First, we compare the similarity of the generated vulnerability fragments to real contract vulnerabilities, and use the highest similarity score as a quantitative metric. The metric threshold is set such that if the generated fragment is highly similar to a real vulnerability, it is simply a repetition of a typical vulnerability and cannot guarantee the diversity of vulnerability fragments, which would render the evaluation of detection tools subjective. Therefore, we choose 60% as the threshold. If the metric exceeds 60%, the generated fragment is considered authentic. The results are shown in Table 6, where each result is presented as <number of actual fragments/total fragments>.

From the experimental results, the SGDL method generates vulnerability fragments that outperform the baseline method SolidiFi in the authenticity comparison. To better understand the above authenticity experimental results and improve the ability to construct smart contract vulnerability fragments, the samples with poor authenticity experimental results were manually verified.

The vulnerability fragments with poor results are roughly divided into two cases: the first corresponds to a pattern of three vulnerability fragments, as shown in Figure 11. The inauthenticity is mainly reflected in 1) the short length of the vulnerability fragment and 2) the fact that the vulnerability fragment contains a single statement component that cannot form a complete function.

An example of the second case is shown in Figure 12. The unauthenticity is mainly reflected in constructing unnatural expressions to construct a specific vulnerability pattern. The "1==1" statement constructed to invalidate the decision statement is unlikely to exist in the natural environment, resulting in an unauthentic vulnerability fragment. Finally, an example of a vulnerability fragment generated using this approach can be seen in Figure 13, containing not only the statements associated with the integer overflow vulnerability but also the associated address-checking statements, which are relatively more authentic.

Answer to RQ3: The vulnerability fragments generated by the *SGDL* method show higher diversity than those generated by the existing method in addition to generating vulnerability fragments more closely resembling actual vulnerability fragments.

```
function func(address to, uint256 amount) public returns (bool success) {
require(to != address(0));
    balance[msg.sender] = balances[msg.sender] - amount;
    balance[to] = balances[to] + amount;
    return true;
}
```

FIGURE 13 Example of SGDL-generated vulnerability fragment

TABLE 7 Results of the detection tool assessment (* indicates that the detection tool could not detect the vulnerability)

Vulnerability type	Methods	Evaluation results									
		Mythril	Slither	SmartCheck	Securify	Oyente	Manticore	Conkas	Confuzzius	TMP	CGE
Integer Overflow	SGDL	31.2%	*	*	*	34.6%	10.0%	70%	66.8%	*	*
	SolidiFi	28.3%	*	*	*	37.7%	9.6%	72%	68%	*	*
Unhandled Exception	SGDL	58.0%	62.0%	22.6%	60.1%	61.5%	*	*	50.7%	*	*
	SolidiFi	50.1%	67.7%	18.8%	58.6%	60.9%	*	*	52%	*	*
Unchecked External Send	SGDL	62.5%	*	*	60.6%	*	*	0.0%	*	*	*
	SolidiFi	65.3%	*	*	61.4%	*	*	0.0%	*	*	*
tx. origin	SGDL	*	65.6%	57.1%	*	*	*	*	*	*	*
	SolidiFi	*	69.8%	61.6%	*	*	*	*	*	*	*
Timestamp Dependency	SGDL	40.6%	72.18%	23.6%	*	42.1%	25.8%	*	*	91.8%	98.4%
	SolidiFi	42.1%	79.78%	26.9%	*	44.8%	27.1%	*	*	93.6%	100%
Reentrancy	SGDL	40.2%	100%	*	21.1%	23.4%	6.6%	71.1%	57.6%	90%	95.1%
	SolidiFi	38.7%	100%	*	20.2%	21.1%	6.4%	70%	58%	95%	100%
Short Address Attack	SGDL	*	*	*	*	*	*	*	*	*	*

4.11 | RQ4: testing tool performance assessment

To answer RQ4, we evaluated the performance of the SGDL-generated smart contract with state-of-the-art vulnerability detection tools. Considering that many detection tools have emerged recently and it is almost impossible to conduct evaluations on all available tools, we selected 10 tools: Mythril,[41] Slither,[42] SmartCheck,[43] Securify,[44] Oyente,[45] Manticore,[46] Conkas, Confuzzius,[47] TMP,[48] and CGE[49] based on the following criteria:

- **Criterion 1.** Its input is Solidity source code.
- **Criterion 2.** It supports a command-line interface so that we can apply it to buggy contracts automatically.
- **Criterion 3.** It is widely used in current research and supports detecting at least two of the seven targeted vulnerability types.

As a comparison method, SolidiFi adopts an indiscriminate vulnerability injection strategy. To ensure the fairness of the experiment, each supported injection vulnerability type, was first selected using our method to perform the injection operation. Then the same set of original contracts were used for vulnerability injection using SolidiFi. The final experimental results are shown in Table 7, where the numerical values in the table represent the proportion of correctly detected vulnerabilities.

From the results in Table 7, it is evident that *SGDL* can generate more vulnerability samples than existing vulnerability detection tools fail to detect. Among the 10 vulnerability detection methods, the types of vulnerabilities generated by *SGDL* surpass those generated by *SolidiFi*, demonstrating the effectiveness of this approach. Analysis of the experimental results reveals that *SGDL* avoids interference from human factors by employing automated operations. Additionally, the diversity and authenticity of vulnerability snippets generated by *SGDL* are superior, and the vulnerabilities produced by *SGDL* are more covert compared to those generated by *SolidiFi*. From Table 6, it can be seen that the vulnerability fragments generated by SolidiFi have low diversity. The number of valid fragments pre-prepared for each vulnerability type in SolidiFi's vulnerability pool is no more than 10, while the number of contracts to be injected with vulnerabilities is 50. It can be deduced that a large amount of duplication is generated during injection, resulting in biased evaluation results. In contrast, SGDL injects state statements and vulnerable functions into a target contract respectively, which is closer to real smart contract vulnerabilities. Therefore, the results obtained by SGDL are more objective and authentic.

Answer to RQ4: *SGDL* enables the generation of a greater number of vulnerability samples that cannot be detected by existing vulnerability detection tools. This, in turn, allows for a more comprehensive and authentic evaluation of these tools.

4.12 | RQ5: activation experiments

To answer RQ5, we explore whether the injected vulnerabilities that cannot be detected by detection tools can be activated during runtime and thus be exploited by malicious attackers. It does not make sense if such injected vulnerability cannot be triggered, since it is useless to attackers and for vulnerability detection. We assess whether a vulnerability that has not been recognized by a tool can be activated in two ways:

(1) For conventional vulnerabilities such as integer overflow, the open-source Solidity smart contract development environment Remix is used to complete the deployment and testing of the vulnerable contract. If an exception can be detected according to the type of vulnerability, the vulnerability is considered successfully activated. (2) For timestamp dependency, a type of vulnerability that requires special conditions, such as the identity of a miner, it can be triggered by combining compiler syntax detection and manual review to determine whether it can be activated. For manual verification, we hired two researchers with 7 years of experience in smart contract development to verify the activation of smart contract vulnerabilities. These researchers manually verified the activation of each smart contract vulnerability generated, and furthermore, their expertise in the smart contract domain highlights their ability to assess vulnerability activation accurately.

The experimental results are shown in Table 8. The experimental results show that nearly all the vulnerabilities generated by the *SGDL* can be activated, in contrast to many non-activatable cases of SolidiFi. After manual review, it can be found that there are two main reasons behind the activation failure of SolidiFi, which are dead code after injection and syntax errors after injection. In our experiments on activating reentrancy vulnerabilities, we observed a significant performance difference between *SGDL* and SolidiFi. This discrepancy primarily stems from the differences in their vulnerability injection mechanisms and considerations of the vulnerability context. *SGDL* employs a more detailed vulnerability generation strategy, not only generating vulnerability code snippets that are closely related to the contract logic but also simulating the context in which the vulnerabilities are triggered, making the generated vulnerabilities more likely to be activated during contract execution. This approach ensures that the injected vulnerabilities exist not only in the code but can also be actually triggered and exploited during the contract's runtime. Overall, the activation rate of the vulnerabilities generated by the *SGDL* method proposed in this paper is higher than that of SolidiFi.

Answer to RQ5: More vulnerabilities generated by *SGDL* can be successfully activated than the existing method.

4.13 | Threat to validity

We acknowledge several potential threats to the validity of our study.

4.13.1 | Internal validity

The potential threats to internal validity originate from the following three aspects:

- *Restricted extensibility*. Up until now, we have utilized *SGDL* to generate the seven types of smart contract vulnerabilities. The implementation of *SGDL* relies on researchers' in-depth understanding of smart contract vulnerabilities. When constructing and labeling the dataset, we manually collected and built a dataset of vulnerable snippets. However, due to subjectivity, such a process may not be objective and may not accurately reflect the distribution of real vulnerabilities. To mitigate this issue, we employed generative adversarial networks (GANs) for

TABLE 8 Activation experiments.

Vulnerability type	SGDL	SolidiFi
Integer Overflow and Underflow	97.9%	97.6%
Unhandled Exception	100%	96.9%
Unchecked External Send	100%	96.3%
Use tx. origin for authentication	100%	82.5%
Timestamp Dependency	98.1%	91.7%
Reentrancy	93.7%	40.5%
Short Address Attack	100%	*

vulnerability snippet generation. This allows us to build an objective and diverse pool of smart contract vulnerability snippets. We also utilized code idioms to improve the quality of the vulnerability snippets during the process.

- *Limited types of supported vulnerabilities*. *SGDL* can currently generate only seven types of vulnerabilities. According to the research,[16] as the number of Ethereum smart contracts increases and the deployment environments evolve, there are more than seven types of smart contract vulnerabilities present on the Ethereum network. Yet, comprehensive datasets for several vulnerabilities are scarce or not openly accessible for academic pursuits. As a result, our focus has remained on a dataset encompassing our chosen seven vulnerabilities. It's important to note that our selection was informed by the prevalence and severity of these vulnerabilities. In our future research, we aim to expand the scope of vulnerability generation to include other types as well.
- *Predefined Vulnerability Categories*. *SGDL* relies on a labeled dataset that defines categories of vulnerabilities. While this approach ensures that the generated vulnerabilities are realistic and meaningful, it also means that SGDL might not generate novel vulnerabilities that fall outside these predefined categories. To mitigate this issue, we expand our dataset by incorporating a broader and more diverse range of vulnerability examples. This expansion will help train our GAN to generate a wider array of vulnerabilities, thereby providing a more robust dataset for evaluating detection tools.

4.13.2 | External validity

There may be potential threats to the external validity stemming from the tools that *SGDL* relies on. *SGDL* utilizes open-source tools, namely *solc* and *ANTLR*, for constructing the control and data flow of contracts. The performance of these tools can also impact the effectiveness of *SGDL*. However, it is worth noting that *solc* and *ANTLR* are widely used open-source tools that are regularly maintained by dedicated organizations and developers. This helps mitigate the impact of this potential threat to a certain extent.

5 | RELATED WORK

5.1 | Vulnerability collection

In response to various computer security incidents, researchers in academia and industry have been working on analyzing security vulnerabilities in applications and developing a variety of detection tools. However, how to evaluate the performance of these tools is a problem worth studying.[50] An ideal evaluation benchmark can identify the blind spots of vulnerability detection tools, improve the performance of detection tools, and broaden their usage scenarios.[51]

Currently, there are three main types of assessment benchmarks constructed, namely, manual construction of program vulnerabilities, collection of real program vulnerabilities, and automated generation of program vulnerabilities based on vulnerability injection techniques. Among them, manual construction of program vulnerabilities and collection of real program vulnerabilities are labor-intensive and difficult to scale up. In comparison, vulnerability injection is less expensive and can easily generate large datasets. In addition, certain vulnerability injection techniques, such as,[52] also enable customization of vulnerability samples to flexibly match various evaluation needs of vulnerability detection tools.

In terms of technical implementation, vulnerability injection can be further divided into two main categories. The first category focuses on finding sensitive locations in the program code through static analysis techniques and constructing evaluation benchmark datasets by injecting contaminated vulnerability fragments in these locations.[53] The second category aims to insert vulnerabilities into the source program by identifying user-controlled inputs that may trigger out-of-bounds reads and writes.[54]

5.2 | Program mutation

Program mutation is a technique in software testing primarily used to assess the quality of a test suite. It involves introducing small changes (known as mutations) to the source code of a program, creating several slightly different versions of the program (mutants). Dong et al[55] proposed the PMTDGM framework, which initially generates mutation-based paths according to the relevance of mutation branches and the difficulty of covering these branches. Subsequently, a multi-task model for path coverage is established. Finally, a Multi-population Genetic Algorithm (MGA) is employed to generate test data. Asghari et al[56] proposed an error propagation-aware mutation testing approach. Tang et al[57] introduced a method named DIPROM, which stands for DIversity-PROMoted mutation, used to construct diversified warning-sensitive programs for effective compiler warning defect detection. Tarimci et al[58] developed a tool called muPLSQL for mutation testing of PL/SQL programs.

In summary, program mutation requires manually defining mutation rules, and the quality and utility of the generated mutants are limited by these rules. Additionally, program mutation relies on predefined mutation operations and can only make local changes to the code. Therefore,

there is a pressing need for an automated vulnerability generation method that does not require manual specification of rules, facilitating developers' early detection and remediation of potential security issues, thereby reducing economic losses.

5.3 | Vulnerability generation

Most current vulnerability generation research is based on vulnerability injection and conducted in traditional programming languages like C/C++ and JAVA. For example, Pewny et al[59] developed a vulnerability injection tool called EvilCoder. This tool uses static analysis techniques to find sensitive code locations that match typical vulnerability patterns and attempts to transform the source code at that location. In the same year, Dolan-Gavitt et al[60] developed LAVA, a tool based on a dynamic taint analysis technique. Unlike EvilCoder, each vulnerability constructed by LAVA is accompanied by an input that triggers it. This input is highly unlikely to exist in normal circumstances. Based on these two tools, Kashyap et al[61] then developed BUG-INJECTOR, which automatically customizes evaluation benchmarks for static analysis tools. The basic idea is to inject a template-based vulnerability into the evaluated detection tool program and dynamically track it by running tests that search for state points that meet specific prerequisites of the vulnerability template and then modify the program being evaluated to inject the vulnerability based on that template. In deep learning, Ahmad et al[62] introduced a unified pre-trained model, PLBART, for program understanding and generation. Liguori et al[63] proposed a method (EVIL) for generating vulnerability code in assembly/Python from natural language descriptions. Compared to those above deep learning-based vulnerability code generation research, our method, optimized for the specific domain, is particularly suited for vulnerability detection and generation in smart contracts, making it outstanding in addressing complex security issues related to smart contracts. Similarly, there lack of vulnerability data in the area of smart contracts. Therefore, Ghaleb et al[11] followed Kashyap et al's idea and proposed SolidiFi, the first smart contract vulnerability injection tool.

However, SolidiFi's vulnerability fragments are manually developed, in contrast to the automated generation of SGDL. In addition, SolidiFi needs to manually specify the type of vulnerability to be injected, compared with SGDL, which can smartly determine the appropriate type of injected vulnerability. These benefits are believed to significantly enhance the productivity and authenticity of vulnerability contract generation.

5.4 | Vulnerability detection

Existing smart contract vulnerability detection tools can further be divided into three categories: symbolic execution, abstract interpretation, and fuzzing. Feist[42] introduced *Slither*, which leverages symbolic execution. This tool initially converts Solidity code into an intermediary representation called SlithIR, utilizing the retained semantic information for vulnerability detection. On the other hand, Tsankov et al[44] developed *Securify* using abstract interpretation. This differs from the vulnerability standards of SWC and *Slither*, as it determines contracts and violation patterns by examining the program's dependency graph, thereby extracting vulnerability semantic information. *SmartCheck*, proposed by Tikhomirov et al,[43] pinpoints contract vulnerabilities by transforming Solidity source code into an XML-based intermediary representation, followed by a comparison with a pre-established XPath path. Luu et al[45] built *Oyente*, which constructs a contract control flow graph at the bytecode level, whereas Torres et al[64] proposed *Osiris*, which formulates basic blocks of contracts with integer overflow errors through CFG analysis also at the bytecode level. Consensys[41] presented *Mythril*, which simulates contract invocations through multiple symbolic executions for vulnerability detection. Torres et al[47] proposed *ConFuzzius*, which is a hybrid test fuzzifier combining evolutionary fuzz testing and constraint solving. The discrepancies between their execution results are eventually analyzed. Additionally, Kalra et al[65] built a framework, *ZEUS*, designed to assess the robustness of smart contracts by employing both symbolic execution and abstract interpretation. Meanwhile, Mossberg et al[46] proposed *Manticore*, a framework for dynamic symbolic execution through analysis of binaries and smart contracts.

In summary, the prevalent paradigm in smart contract vulnerability detection is predominantly centered on human specialists formulating detection regulations throughout the design phase. Such an approach grapples with challenges including limited scalability, diminished precision, and elevated expenses. As a consequence, there emerges a pressing necessity to amass an extensive dataset comprising a spectrum of vulnerability types, facilitating developers in the assessment of their tool's efficacy.

6 | CONCLUSION AND FUTURE WORK

In this paper, we aim to generate more authentic and diverse smart contract vulnerabilities through the powerful data-fitting capabilities of generative adversarial networks. We obtain positive findings in improving the diversity and realism of exploit fragments while ensuring the syntactic correctness of the exploit-bearing smart contracts. Extensive experimental results show that the quality, diversity, and validity of the vulnerability fragments generated by *SGDL* outperform the existing vulnerability injection method. Notably, in terms of diversity, it showcases an average

improvement of 69.99% when compared to existing methods. Furthermore, in terms of authenticity, it achieves an average improvement of 62.2%. We believe that *SGDL* is an important step forward in deep learning-based vulnerability injection for smart contracts.

For future work, we will develop additional data collection rules to support more vulnerability types and construct datasets that cover a wider range of vulnerability types. Additionally, we will strive to improve the structure of the generative adversarial network model to generate more realistic smart contract vulnerability snippets. We will also explore the possibility of incorporating other fine-grained information into *SGDL*.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.

ORCID

Hanting Chu https://orcid.org/0009-0005-4412-9871
Shunhui Ji https://orcid.org/0000-0002-8584-5795

ENDNOTES

* https://dasp.co/#item-9

† https://github.com/Tourneso/SGDL

‡ DASP. https://www.dasp.co

§ ETH DApp attacks (https://hacked.slowmist.io/?c=ETH)

¶ https://gitter.im/orgs/ethereum/rooms/

https://github.com/cleanunicorn/karl

** https://etherscan.io

†† https://github.com/solidity-parser/antlr

REFERENCES

1. Wang Z, Jin H, Dai W, Choo KKR, Zou D. Ethereum smart contract security research: survey and future research opportunities. *Front Comp Sci*. 2021; 15(2):152802. doi:10.1007/s11704-020-9284-9
2. Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T. Defining smart contract defects on ethereum. *IEEE Trans Softw Eng*. 2020:327-345.
3. Hewa T, Ylianttila M, Liyanage M. Survey on blockchain based smart contracts: applications, opportunities and challenges. *J Netw Comput Appl*. 2021; 177:102857. doi:10.1016/j.jnca.2020.102857
4. Cai J, Li B, Zhang J, Sun X. Ponzi scheme detection in smart contract via transaction semantic representation learning. *IEEE Trans Reliab*. 2023;73(2): 1117-1131. doi:10.1109/TR.2023.3319318
5. Kushwaha SS, Joshi S, Singh D, Kaur M, Lee HN. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*. 2022;10:6605-6621. doi:10.1109/ACCESS.2021.3140091
6. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok). In: *International conference on principles of security and trust springer*. Springer Berlin Heidelberg; 2017:164-186. doi:10.1007/978-3-662-54455-6_8
7. Cai J, Li B, Zhang J, Sun X, Chen B. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *J Syst Softw*. 2023;195:111550. doi:10.1016/j.jss.2022.111550
8. Sayeed S, Marco-Gisbert H, Caira T. Smart contract: attacks and protections. *IEEE Access*. 2020;8:24416-24427. doi:10.1109/ACCESS.2020.2970495
9. Zou W, Lo D, Kochhar PS, et al. Smart contract development: challenges and opportunities. *IEEE Trans Softw Eng*. 2019;47(10):2084-2106. doi:10.1109/TSE.2019.2942301
10. Ren X, Wu Y, Li J, Hao D, Alam M. Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network. *Comput Electr Eng*. 2023;109:108766. doi:10.1016/j.compeleceng.2023.108766
11. Ghaleb A, Pattabiraman K. 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; 2020. p. 415-427.
12. Zhang J, Wang X, Zhang H, Sun H, Wang K, Liu X. A novel neural source code representation based on abstract syntax tree. In: *2019 IEEE/ACM 41st international conference on software engineering (ICSE)*. IEEE; 2019:783-794.
13. Neamtiu I, Foster JS, Hicks M. Understanding source code evolution using abstract syntax tree matching. In: *Proceedings of the 2005 international workshop on mining software repositories*; 2005:1-5.
14. Cui B, Li J, Guo T, Wang J, Ma D. Code comparison system based on abstract syntax tree. In: *2010 3rd IEEE international conference on broadband network and multimedia technology (IC-BNMT)*. IEEE; 2010:668-673.
15. Dingman W, Cohen A, Ferrara N, et al. Classification of smart contract bugs using the nist bugs framework. In: *2019 IEEE 17th international conference on software engineering research, management and applications (SERA)*. IEEE; 2019:116-123.
16. Zhang P, Xiao F, Luo X. A framework and dataset for bugs in ethereum smart contracts. In: *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE; 2020:139-150.
17. Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B. Reguard: finding reentrancy bugs in smart contracts. In: *2018 IEEE/ACM 40th international conference on software engineering: companion (ICSE-companion)*. IEEE; 2018:65-68.

18. Huang Y, Bian Y, Li R, Zhao JL, Shi P. Smart contract security: a software lifecycle perspective. *IEEE Access*. 2019;7:150184-150202. doi:10.1109/ACCESS.2019.2946988
19. Parizi RM, Dehghantanha A, Choo KKR, Singh A. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. arXiv preprint arXiv:180902702 2018.
20. Durieux T, Ferreira JF, Abreu R, Cruz P. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering; 2020. p. 530-541.
21. Buterin V. A next-generation smart contract and decentralized application platform. *White Paper*. 2014;3(37):28-34.
22. Wang W, Song J, Xu G, Li Y, Wang H, Su C. Contractward: automated vulnerability detection models for ethereum smart contracts. *IEEE Trans Netw Sci Eng*. 2020;8(2):1133-1144. doi:10.1109/TNSE.2020.2968505
23. Delmolino K, Arnett M, Kosba A, Miller A, Shi E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: *International conference on financial cryptography and data security*. Springer; 2016:79-94. doi:10.1007/978-3-662-53357-4_6
24. Perez AJ, Zeadally S. Secure and privacy-preserving crowdsensing using smart contracts: issues and solutions. *Comput Sci Rev*. 2022;43:100450. doi:10.1016/j.cosrev.2021.100450
25. Harz D, Knottenbelt W. Towards safer smart contracts: A survey of languages and verification methods. arXiv preprint arXiv:180909805 2018.
26. Sürücü O, Yeprem U, Wilkinson C, et al. A survey on ethereum smart contract vulnerability detection using machine learning. Disruptive Technologies in Information Sciences VI 2022;12117:110-121.
27. Chen J, Xia X, Lo D, Grundy J, Yang X. Maintenance-related concerns for post-deployed Ethereum smart contract development: issues, techniques, and future challenges. *Empir Softw Eng*. 2021;26(6):117. doi:10.1007/s10664-021-10018-0
28. Wang S, Yuan Y, Wang X, Li J, Qin R, Wang FY. An overview of smart contract: architecture, applications, and future trends. In: *2018 IEEE intelligent vehicles symposium (IV)*. IEEE; 2018:108-113.
29. Liao M, Shi B, Bai X, Wang X, Liu W. Textboxes: A fast text detector with a single deep neural network. In: Proceedings of the AAAI conference on artificial intelligence, vol. 31; 2017. .
30. Shin EC, Allamanis M, Brockschmidt M, Polozov A. Program synthesis and semantic parsing with learned code idioms. *Adv Neural Inf Process Syst*. 2019;32:10825-10835.
31. Guo J, Lu S, Cai H, Zhang W, Yu Y, Wang J. Long text generation via adversarial training with leaked information. In: Proceedings of the AAAI conference on artificial intelligence, vol. 32; 2018. .
32. Theis L, Oord Avd, Bethge M. A note on the evaluation of generative models. arXiv preprint arXiv:151101844 2015.
33. Semeniuta S, Severyn A, Gelly S. On accurate evaluation of gans for language generation. arXiv preprint arXiv:180604936 2018.
34. Huszár F. How (not) to train your generative model: Scheduled sampling, likelihood, adversary? arXiv preprint arXiv:151105101 2015.
35. Papineni K, Roukos S, Ward T, Zhu WJ. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th annual meeting of the Association for Computational Linguistics; 2002. p. 311-318.
36. Yu L, Zhang W, Wang J, Yu Y. Seqgan: Sequence generative adversarial nets with policy gradient. In: Proceedings of the AAAI conference on artificial intelligence, vol. 31; 2017. .
37. Lin K, Li D, He X, Zhang Z, Sun MT. Adversarial ranking for language generation. *Adv Neural Inf Process Syst*. 2017;30:3158-3168.
38. Cherkassky V, Ma Y. Practical selection of SVM parameters and noise estimation for SVM regression. *Neural Netw*. 2004;17(1):113-126. doi:10.1016/S0893-6080(03)00169-2
39. Guo G, Wang H, Bell D, Bi Y, Greer K. KNN model-based approach in classification. In: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings Springer; 2003. p. 986-996, doi:10.1007/978-3-540-39964-3_62.
40. Fan J, Ma X, Wu L, Zhang F, Yu X, Zeng W. Light gradient boosting machine: an efficient soft computing model for estimating daily reference evapotranspiration with local and external meteorological data. *Agric Water Manag*. 2019;225:105758. doi:10.1016/j.agwat.2019.105758
41. Mueller B. *Smashing ethereum smart contracts for fun and real profit*. Vol. 9. HITB SECCONF Amsterdam; 2018:54.
42. Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In: *2019 IEEE/ACM 2nd international workshop on emerging trends in software engineering for Blockchain (WETSEB)*. IEEE; 2019:8-15.
43. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain; 2018. p. 9-16.
44. Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Buenzli F, Vechev M. Securify: Practical security analysis of smart contracts. In: *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. ACM; 2018:16-26.
45. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security; 2016. p. 254-269.
46. Mossberg M, Manzano F, Hennenfent E, et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: *2019 34th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE; 2019:1186-1189.
47. Torres CF, Iannillo AK, Gervais A, State R. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In: *2021 IEEE European symposium on security and privacy (EuroS&P)*. IEEE; 2021:103-119.
48. Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q. *Smart contract vulnerability detection using graph neural network*. IJCAI; 2020:3283-3290.
49. Liu Z, Qian P, Wang X, Zhuang Y, Qiu L, Wang X. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Trans Knowl Data Eng*. 2021;1. doi:10.1109/TKDE.2021.3095196
50. Wan Z, Xia X, Lo D, Chen J, Luo X, Yang X. Smart contract security: a practitioners' perspective. In: *2021 IEEE/ACM 43rd international conference on software engineering (ICSE)*. IEEE; 2021:1410-1422.
51. Jin H, Wang Z, Wen M, Dai W, Zhu Y, Zou D. Aroc: an automatic repair framework for on-chain smart contracts. *IEEE Trans Softw Eng*. 2021;48(11):4611-4629. doi:10.1109/TSE.2021.3123170
52. Kindy DA, Pathan ASK. A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques. In: *2011 IEEE 15th international symposium on consumer electronics (ISCE)*. IEEE; 2011:468-471.
53. Antunes J, Neves N, Correia M, Verissimo P, Neves R. Vulnerability discovery with attack injection. *IEEE Trans Softw Eng*. 2010;36(3):357-370. doi:10.1109/TSE.2009.91

54. Chen JM, Wu CL. An automated vulnerability scanner for injection attack based on injection point. In: *2010 international computer symposium (ICS2010)*. IEEE; 2010:113-118.
55. Dang X, Wang J, Gong D, Yao X, Wei C, Xu B. Test data generation for covering mutation-based path using MGA for MPI program. *J Syst Softw*. 2024; 210:111962. doi:10.1016/j.jss.2024.111962
56. Asghari Z, Arasteh B, Koochari A. Effective software mutation-test using program instructions classification. *J Electron Test*. 2023;39(5):631-657. doi: 10.1007/s10836-023-06089-0
57. Tang Y, Jiang H, Zhou Z, Li X, Ren Z, Kong W. Detecting compiler warning defects via diversity-guided program mutation. *IEEE Trans Software Eng*. 2022;48(11):4411-4432. doi:10.1109/TSE.2021.3119186
58. Tarimci AB, Sözer H. Mutation testing of PL/SQL programs. *J Syst Softw*. 2022;192:111399. doi:10.1016/j.jss.2022.111399
59. Pewny J, Holz T. EvilCoder: automated bug insertion. In: Proceedings of the 32nd Annual Conference on Computer Security Applications; 2016. p. 214-225.
60. Dolan-Gavitt B, Hulin P, Kirda E, et al. Lava: Large-scale automated vulnerability addition. In: *2016 IEEE symposium on security and privacy (SP)*. IEEE; 2016:110-121.
61. Kashyap V, Ruchti J, Kot L, et al. Automated customized bug-benchmark generation. In: *2019 19th international working conference on source code analysis and manipulation (SCAM)*. IEEE; 2019:103-114.
62. Ahmad WU, Chakraborty S, Ray B, Chang KW. Unified pre-training for program understanding and generation. arXiv preprint arXiv:210306333 2021.
63. Liguori P, Al-Hossami E, Orbinato V, et al. EVIL: exploiting software via natural language. In: *2021 IEEE 32nd international symposium on software reliability engineering (ISSRE)*. IEEE; 2021:321-332.
64. Torres CF, Schütte J, State R. Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference; 2018. p. 664-676.
65. Kalra S, Goel S, Dhawan M, Sharma S. *Zeus: analyzing safety of smart contracts*. Ndss; 2018:1-12.

How to cite this article: Chu H, Zhang P, Dong H, Xiao Y, Ji S. *SGDL*: Smart contract vulnerability generation via deep learning. *J Softw Evol Proc*. 2024;36(12):e2712. https://doi.org/10.1002/smr.2712