sGuard+: Machine Learning Guided Rule-Based Automated Vulnerability Repair on Smart Contracts

CUIFENG GAO, School of Computer Science and Technology, University of Science and Technology of China, Hefei, China and Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, China
WENZHANG YANG, University of Science and Technology of China, Hefei, China
JIAMING YE, Kyushu University, Fukuoka, Japan
YINXING XUE, School of Computer Science and Technology, University of Science and Technology of China, Hefei, China and Suzhou Institute for Advanced Research, University of Science and Technology of China, Suzhou, China
JUN SUN, Singapore Management University, Singapore, Singapore

Smart contracts are becoming appealing targets for hackers because of the vast amount of cryptocurrencies under their control. Asset loss due to the exploitation of smart contract codes has increased significantly in recent years. To guarantee that smart contracts are vulnerability-free, there are many works to detect the vulnerabilities of smart contracts, but only a few vulnerability repair works have been proposed. Repairing smart contract vulnerabilities at the source code level is attractive as it is transparent to users, whereas existing repair tools, such as SCRepair and sGuard, suffer from many limitations: (1) ignoring the code of vulnerability prevention; (2) possibly applying the repair to the wrong statements and changing the original business logic of smart contracts; and (3) showing poor performance in terms of time and gas overhead.

In this work, we propose machine learning guided rule-based automated vulnerability repair on smart contracts to improve the effectiveness and efficiency of sGuard. To address the limitations mentioned above, we design the features that characterize both the symptoms of vulnerabilities and the methods of vulnerability prevention to learn various vulnerability patterns and reduce false positives. Additionally, a fine-grained localization algorithm is designed by traversing the nodes of the abstract syntax tree, and we refine and extend the repair rules of sGuard to preserve the original business logic of smart contracts and support new

This work was supported in part by the Anhui Provincial Department of Science and Technology under Grant 202103a05020009, in part by National Natural Science Foundation of China under Grant 61972373, the Basic Research Program of Jiangsu Province under Grant BK20201192. The research of Dr. Xue is also supported by CAS Pioneer Hundred Talents Program of China. Jun Sun's research was supported in part by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.
Authors' addresses: C. Gao and Y. Xue (Corresponding author), School of Computer Science and Technology, University of Science and Technology of China, 96 Jinzhai Road, Baohe District, Hefei City, Anhui Province, China, 230026, and Suzhou Institute for Advanced Research, University of Science and Technology of China, 188 Renai Road, Suzhou City, Jiangsu Province, China, 215123; e-mails: gcf20225162@mail.ustc.edu.cn, yxxue@ustc.edu.cn; W. Yang, University of Science and Technology of China, 96 Jinzhai Road, Baohe District, Hefei City, Anhui Province, China, 230026; e-mail: yywwzz@mail.ustc.edu.cn; J. Ye, Kyushu University, 744 Motooka Nishi-ku, Fukuoka, Japan, 819-0395; e-mail: yejjmg@gmail.com; J. Sun, Singapore Management University, 81 Victoria Street, Singapore, 188065; e-mail: junsun@smu.edu.sg.

ACM 1049-331X/2024/06-ART114
https://doi.org/10.1145/3641846

vulnerability types. Our tool, named sGuard+, reduces time overhead based on machine learning models, and reduces gas overhead by fewer code changes and precise patching.

In our experiment, we collect a publicly available vulnerability dataset from CVE, SWC, and SmartBugs Curated as a ground truth for evaluations. Overall, sGuard+ repairs more vulnerabilities with less time and gas overhead than state-of-the-art tools. Furthermore, we reproduce about 9,000 historical transactions for regression testing. It is shown that sGuard+ has no impact on the original business logic of smart contracts.

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

Additional Key Words and Phrases: Vulnerability repair, smart contract, machine learning

ACM Reference Format:
Cuifeng Gao, Wenzhang Yang, Jiaming Ye, Yinxing Xue, and Jun Sun. 2024. sGuard+: Machine Learning Guided Rule-Based Automated Vulnerability Repair on Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 33, 5, Article 114 (June 2024), 55 pages. https://doi.org/10.1145/3641846

1 INTRODUCTION

Ethereum is one of the most popular blockchain platforms [56]. It is designed to enable self-enforcing programs, called smart contracts, to execute transactions automatically. In recent years, thousands of applications built based on smart contracts have revolutionized many areas [34], such as finance, arts and collectibles, gaming, and technology. Unlike traditional computer programs, smart contracts can process not only data but also cryptocurrencies, so that a large amount of digital assets are under the control of smart contracts. With the rise of the market capitalization of cryptocurrencies, smart contracts have become appealing targets for hackers. For instance, the notorious DAO hack [43] resulted in 70 million dollars being plundered in 2016, and the same vulnerability was exploited by the Uniswap/Lendf.Me hack [82] in 2020 and the Cream Finance hack [85] in 2021. The Crypto Crime Report [18] released by Chainalysis shows that more than 2 billion dollars are stolen in the finance area in 2021. Among them, the loss caused by smart contract code exploitation accounts for approximately half. To protect property safety and maintain social stability, it is essential and urgent to ensure that smart contracts are vulnerable-free.

Vulnerability detection and repair are integral parts of the smart contract development life cycle. In recent years, there have been a large number of works on detecting different vulnerabilities [28], whereas only a few vulnerability repair works have been proposed to guarantee that the vulnerabilities are free. To the best of our knowledge, there are only two automatic vulnerability repair tools SCRepair [115] and sGuard [75] that work at the source code level of smart contracts. SCRepair adopts a gas-aware genetic search technique that generates candidate patches using a set of predefined mutation operators and chooses patches with low gas consumption. However, the search-based approach applied by SCRepair has limited effectiveness with respect to vulnerabilities that require complex code changes [42]. In contrast, sGuard adopts a rule-based approach that has the advantage of defining complex change rules to repair different vulnerabilities. However, the effectiveness and efficiency of sGuard need to be improved due to the following limitations.

- L1 *The effectiveness of vulnerability detection.* sGuard Performs static analysis on symbolic execution traces to detect vulnerabilities. Similar to existing static analysis tools [32], sGuard suffers from a high false positive rate. As a result, the patches introduced by sGuard due to false positives cause a large amount of unnecessary gas overhead, so that the transaction fees of the repaired smart contracts increase significantly.
- L2 *The correctness of vulnerability repair.* Correctly repairing a smart contract requires us to remove the vulnerability whilst preserving the business logic. However, the repair rules of

sGuard may change the original business logic when the vulnerable code has interprocedural calls between functions or side effects of expressions. One of the repair rules of sGuard even introduces a new access control vulnerability CVE-2020-19765 [3]. Moreover, sGuard implements code transformations at the string level rather than the **abstract syntax tree** (**AST**), which sometimes leads to uncompilable contracts.

L3 *The time and gas overhead.* As shown in the experiments of sGuard, more than one third of smart contracts are not repaired due to timeout. Therefore, the time consumption of sGuard can be greatly improved as the complexity of smart contracts increases. Moreover, the redundant patch code introduced by sGuard significantly increases the gas overhead.

In this work, we propose a machine learning-guided automated vulnerability repair approach, which aims to improve the effectiveness and efficiency of sGuard with respect to both vulnerability detection and repair. To address the limitation L1, we utilize the machine learning technique to learn vulnerability patterns in a general way without being limited by the specific testing and analysis approaches applied by rule-based tools. In particular, we design not only the features (Vul-Fs) that characterize the symptoms of vulnerabilities, but also the features (Pre-Fs) that characterize the methods of vulnerability prevention, as is known in secure coding practices [68], to reduce false positives. To address limitation L2, we refine and extend the repair rules of sGuard to preserve the original business logic of smart contracts, and the code transformation is implemented based on AST nodes to guarantee that the syntax of the repaired smart contracts is correct. To address limitation L3, we design new repair rules that require fewer code changes and reduce unnecessary patch code based on accurate localization to reduce gas overhead. Furthermore, the time performance of the machine learning models outperforms sGuard.

In this work, we implement a tool, named sGuard+, which works by the following three key steps.

S1 *Vulnerability detection.* Binary classification machine learning models are utilized to detect each vulnerability type at the granularity of the function level. The training set is labeled according to three state-of-the-art vulnerability detection tools (Slither, Securify, and Mythril). The features that contain Vul-Fs and Pre-Fs are extracted from both the source code and the bytecode of smart contracts.

S2 *Vulnerability localization.* A localization algorithm is designed to identify candidate statements for repair in the vulnerable functions reported by machine learning models. Specifically, the AST nodes of each vulnerable function are traversed to find the target statements or variables that need to be modified. Note that this step can be used to further filter false positives that cannot find the target code.

S3 *Vulnerability repair.* The localized vulnerable AST nodes are transformed according to the repair rules designed for each specific type of vulnerability. To preserve the original business logic of smart contracts, we refine and extend the repair rules of sGuard by defining more precise conditions based on the analysis of the **interprocedural call graph** (**ICG**) and side effects of expressions. Additionally, new repair rules are designed for two new types of vulnerability and fewer gas overhead.

In our experiment, the **eXtreme Gradient Boosting** (**XGBT**) model for binary classification is selected because it has the highest average F1 score of 0.78 on the training set and 0.86 on the test set. Its F1 score for each vulnerability type is close to that of the best performing tool among the three state-of-the-art rule-based tools, and significantly higher than the results of sGuard and the two existing ML-based tools. Additionally, we collect a publicly available vulnerability dataset from CVE [27], SWCRegistry [72], and SmartBugs Curated ($SB^{CURATED}$) [32] as a ground truth of vulnerabilities to evaluate the effectiveness and efficiency of sGuard+. Overall, the

vulnerability repair rate of sGuard+ increased by 51 pp over sGuard on PVD, and sGuard+ preserves the original business logic according to regression testing. sGuard+ only takes approximately one-fifth the time of sGuard, and the gas overhead of deploying contracts and calling functions of sGuard+ decreased 7.1 pp and 6.1 pp, respectively, compared with sGuard. Furthermore, we evaluate sGuard+ in the dataset of the experiment of SCRepair so that sGuard+ can be compared with SCRepair based on the results shown in the article of SCRepair. The results show that the vulnerability repair rate of sGuard+ increased by 44 pp over SCRepair, and sGuard+ consumes only 0.3% of the time spent by SCRepair.

To sum up, the contributions of this work are listed as follows.

— Our work makes the first attempt to use machine learning to guide automated vulnerability repair at the source code level of smart contracts to improve the effectiveness and efficiency of the repair tool.
— We design the features that characterize both the symptoms of vulnerabilities and the methods of vulnerability prevention to learn various vulnerability patterns and reduce false positives.
— We extend and improve the repair rules of sGuard to further guarantee its correctness at the AST level. Our repair rules are evaluated by large-scale regression testing with thousands of historical transactions, which shows that sGuard+ successfully repairs sGuard's unintended modification of the business logic.
— We implement the tool sGuard+ that supports five types of vulnerability and compare sGuard+ with state-of-the-art smart contract repair tools (sGuard and SCRepair) at the source code level. The results show that sGuard+ repairs more vulnerabilities than sGuard and SCRepair with less time and gas overhead. The source code of sGuard+ and the datasets are publicly accessible.[1]

2 BACKGROUND

In this section, we first introduce smart contracts and `Solidity`. Then, we present five types of common vulnerabilities of smart contracts. Finally, using a motivating example, we discuss the challenges of automatically repairing or fixing these vulnerabilities using state-of-the-art tools.

2.1 Smart Contracts and Solidity Language

A smart contract can be seen as a vending machine, as described by Nick Szabo [98], which guarantees a certain output with the right inputs. Since smart contracts hold substantial funds and react to transactions sent by executing code, malicious attacks exploiting program vulnerabilities (e.g., DAO [1] and Parity wallet hack [2]) have caused huge economic losses. In this work, we focus on smart contracts on Ethereum, which is the most popular blockchain platform and supports Turing-complete programming languages for smart contracts.

In Ethereum, users can create two types of accounts: the **external owned account** (**EOA**) controlled by private keys, and the contract account controlled by smart contract code. Both account types can store and transfer Ethereum's native cryptocurrency, named Ether, and exist in the form of a 42-character hexadecimal address. The only way for users to update the state of the Ethereum network is to initiate a transaction with the EOA. A *transaction* can be a direct ether transfer from one EOA to another EOA, the deployment of a contract, or the execution of a contract code. Every transaction must be verified by the miners before being recorded on the blockchain. However, in order to prevent malicious transactions from executing code with an infinite loop to waste miner's

[1] https://doi.org/10.5281/zenodo.8249340

computing resources, users need to pay enough additional fees, called *gas*, based on the computation consumed by the code. If the transaction does not pay enough gas, it will never be verified successfully. It can be seen that a key difference between smart contracts and traditional programs is that the more complex the contract code, the more gas the user needs to bear.

`Solidity` is the most popular high-level language for implementing smart contracts on Ethereum. `Solidity` is influenced by existing programming languages (e.g., `C++`, `Python` and `JavaScript`), and it has some contract-specific features, such as modifier clauses, event notifiers for listeners, and custom global variables [6]. In general, just like the `class` keyword in `C++`, the `contract` keyword can be used to define a contract object in `Solidity`, which also follows multiple inheritance, including polymorphism. In a contract object, the `function` keyword can be used to define custom-specific functions, and the dot (.) operator can be used to invoke the function associated with the particular object. By default, a contract object has a special function without a function name, called *fallback* function, which cannot accept arguments and return anything. The fallback function is executed when a called function identifier does not match any of the available functions in the contract, or an external call does not supply any data such as whenever the contract receives Ether without any other data associated with the transaction.

In order to interact with data on the blockchain, `Solidity` has many built-in global variables and functions, which can be used to read or update the status of the Ethereum network, or identify the transaction information. For example, variables `tx.origin` and `msg.sender` of the `address` type represent the sender of the transaction and the sender of the message, respectively. Note that these two variables may represent different information because a single transaction may execute multiple contract codes with different addresses. For example, when a EOA $address_{EOA}$ calls a function f_a in the contract address $address_A$, and the function f_a executes the code $address_B.f_b()$, named external call, calling the function f_b in contract address $address_B$. In function f_a, both the variable `tx.origin` and `msg.sender` represent the address $address_{EOA}$, while in function f_b, the variable `tx.origin` also represents the address $address_{EOA}$, but the variable `msg.sender` represents the address $address_A$. Moreover, the built-in function `selfdestruct` can be used to destroy the current contract, and three built-in member functions (`call.value`, `send`, `transfer`) of the `address` type can be used to transfer Ether to an address in the form of `address.transfer()`.

Furthermore, the **Ethereum Virtual Machine** (**EVM**) is a runtime environment for Ethereum transaction execution, and smart contracts developed in `Solidity` are compiled into bytecode that can be executed by the EVM. The bytecode is a binary string, and each byte is an opcode. The EVM opcodes described in the Ethereum Yellow Paper [36] are shown in Table 1. In detail, the EVM opcodes can be roughly divided into 5 groups: (1) The opcodes from 0x00 to 0x20 are related to arithmetic operations. (2) The opcodes from 0x30 to 0x48 are related to environmental information and block information. (3) The opcodes from 0x50 to 0x9f are related to data handling. Note that EVM is stack-based and has three data location types, namely stack, memory, and storage. (4) The opcodes from 0xa0 to 0xa4 are related to logging operations. and (5) The opcodes from 0xf0 to 0xff are related to system operations. Compared with common arithmetic, stack, and log operations, opcodes that obtain key information or refer to critical system operations are often associated with vulnerabilities. For example, the opcode `ORIGIN` for obtaining the execution origination address is commonly used to authorize transactions, which is associated with authorization through the tx-origin vulnerability [104]. The opcode `CALL` to send messages to an account is executed in every transaction sending Ether, which is an essential feature for reentrancy vulnerability and unchecked call return value vulnerability. The opcode `SELFDESTRUCT` to halt execution and register accounts for later deletion means that the contract will be destroyed, which is related to the vulnerability of the unprotected `SELFDESTRUCT` instruction.

Table 1. The Valid Ethereum Virtual Machine Opcodes from Yellow Paper

Bytecodes	Opcodes	Description
0x00 ~ 0x0b	STOP ADD MUL SUB DIV SDIV MOD SMOD ADDMOD MULMOD EXP SIGNEXTEND	STOP and aritmetic operations.
0x10 ~ 0x1d	LT GT SLT SGT EQ ISZERO AND OR XOR NOT BYTE SHL SHR SAR	Comparision and bitwise logic operations.
0x20	SHA3	Compute Keccak-256 hash.
0x30 ~ 0x3f	ADDRESS BALANCE ORIGIN CALLER CALLVALUE CALLDATALOAD CALLDATASIZE CALLDATACOPY CODESIZE CODECOPY GASPRICE EXTCODESIZE EXTCODECOPY RETURNDATACOPY RETURNDATASIZE EXTCODEHASH	Get environmental information.
0x40 ~ 0x48	BLOCKHASH COINBASE TIMESTAMP NUMBER DIFFICULTY GASLIMIT CHAINID SELFBALANCE BASEFEE	Get block information.
0x50 ~ 0x5b	POP MLOAD MSTORE MSTORE8 SLOAD SSTORE JUMP JUMPI PC MISIZE GAS JUMPDEST	Stack, memory, storage, and flow operations.
0x60 ~ 0x7f	PUSH1 ... PUSH32	Push operations.
0x80 ~ 0x8f	DUP1 ... DUP16	Duplication operations.
0x90 ~ 0x9f	SWAP1 ... SWAP16	Exchange operations.
0xa0 ~ 0xa4	Log0 ... LOG4	Logging operations.
0xf0 ~ 0xff	CREATE CALL CALLCODE RETRUN DELEGATECALL CREATE2 STATICCALL REVERT SELFDESTRUCT	System operations.

2.2 Commonly Known Vulnerabilities And Preventative Strategies

Smart Contract Weakness Classification (**SWC**), as a well-known and well-defined standard, was proposed formally in EIP-1470 [105], which aims to classify and identify weaknesses that lead to vulnerability in smart contracts. In this article, we focus on the following five common vulnerabilities.[2]

(I) SWC-101: Integer Overflow and Underflow Vulnerability (IOU)
Vulnerability: A smart contract suffers from integer overflow and underflow vulnerability when an arithmetic operation reaches the maximum or minimum value. The main reason for this vulnerability is that the EVM specifies fixed-size data types for integers and ignores whether the data are outside the range of the variable type. For example, Figure 1(a) shows an example of arithmetic vulnerability, in which the product of `numTokens * PRICE_PER_TOKEN` in line 4 may exceed the range of the data type `uint256`. This integer overflow results in an unexpected value of transfer. Note that although integer overflow and underflow vulnerability is a solved issue with latest `Solidity` compiler [7], it is still relevant since majority of the existing smart contracts still suffer from this problem.
Preventative Strategy: The most common way to avoid the integer overflow and underflow vulnerability is to replace arithmetic operators with corresponding safe math functions

[2]In order to reduce the confusion with the vulnerability definitions in existing papers (e.g., Refs. [20, 75, 102]), we uniformly describe vulnerabilities according to the definition of SWC.

```
1 function sell(uint256 numTokens) public {
2   require(balanceOf[msg.sender] >= numTokens
        );
3   balanceOf[msg.sender] -= numTokens;
4   msg.sender.transfer(numTokens *
        PRICE_PER_TOKEN);
5 }
```

(a) An example of IOU vulnerability from SWC-101 test cases.

```
1 function callnotchecked(address payable
      _callee, uint amount) public {
2   require(balances[_callee] >= amount);
3
4   _callee.call.value(amount)("");
5   balances[_callee] -= amount;
6 }
```

(b) An example of UCR vulnerability from SWC-104 test cases.

```
1 function sudicideAnyone() {
2
3     selfdestruct(msg.sender);
4
5
6 }
```

(c) An example of USI vulnerability from SWC-106 test cases.

```
1 function withdraw(uint amount) public{
2     if (credit[msg.sender]>= amount) {
3         require(msg.sender.call.value(amount
              )());
4         credit[msg.sender]-=amount;}
5 }
```

(d) An example of REN vulnerability from SWC-107 test cases.

```
1 function sendTo(address receiver, uint amount) public {
2   require(tx.origin == owner);
3   receiver.transfer(amount);
4 }
```

(e) An example of TXO vulnerability from SWC-115 test cases.

Fig. 1. Examples of 5 vulnerability types.

[80], which set up safety condition checks for arithmetic operations such as adding the statement `Require((numTokens * PRICE_PER_TOKEN)/numTokens == PRICE_PER_TOKEN);` after the multiplication in line 4 of Figure 1(a) to guarantee the product is correct. If the safety check fails, the transaction executing the current code fails.

(II) SWC-104: Unchecked Call Return Value Vulnerability (UCR)

Vulnerability: A smart contract suffers from unchecked call return value vulnerability if the return value of a low-level call method (e.g., `address.send()`, `address.call()`, `address.delegatecall()` or `address.callcode()`) is unchecked. Each low-level call method returns a value of type Boolean, which indicates that the call is succeeded or failed. Therefore, if the return value is not checked, it will lead to unexpected behavior and change the program's business logic. Figure 1(b) shows an example in which the balance of the address `_callee` will be reduced even if the transaction sending Ether at line 4 fails.

Preventative Strategy: The unchecked call return value vulnerability can be avoided by directly adding a conditional check on the return value. For example, we can first declare a variable of type `Boolean` to store the return value of the low-level call in the forth line of Figure 1(b), such as `bool success = _callee. call.value(amount)("");`, then add the statement `Require(success);` next to the call statement to guarantee that the return value must be true.

(III) SWC-106: Unprotected `SELFDESTRUCT` Instruction Vulnerability (USI)

Vulnerability: A smart contract suffers from unprotected `SELFDESTRUCT` instruction vulnerability when missing or insufficient access controls exist for the `SELFDESTRUCT` instruction. This vulnerability is caused by not checking the identity or permission of the caller when destructing the contract. An example of this vulnerability is shown in Figure 1(c). Any user can destroy the contract and transfer all the Ether by calling this public function since there is no permission check.

Preventative Strategy: The unprotected `SELFDESTRUCT` instruction vulnerability can be avoid by using proper access control on the self-destruct function. As shown in the

secure coding practices for smart contract access control [78], the concept of ownership is the most common and basic form of access control, which only grants the contract owner the permission to call some functions. For example, in Figure 1(c), we can add the statement `Require(msg.sender == _owner);` to prevent any other normal user from calling the function.

(IV) SWC-107: Reentrancy Vulnerability (REN)

Vulnerability: A smart contract suffers from reentrancy vulnerability when a malicious contract calls back into the calling contract in undesirable ways before the first invocation of the function is finished. As shown in Figure 1(d), the business logic of the function `withdraw` is that if the balance (`credit[]`) of the user (`msg.sender`) is greater than the requested amount (line 2), send Ether to the user without any data attached (line 3) and then deduct the user's balance (line 4). It suffers from REN because if the variable `msg.sender` is the address of a malicious contract (C_m) whose self-defined fallback function (f_m) has an external call to this victim function `withdraw` (f_v), sending Ether to the C_m (line 3) will cause the f_m to be executed and then call the f_v again, so that the attacker can withdraw Ether recursively before deducting the transfer amount from its balance.

Preventative Strategy: There are two most common ways to avoid reentrancy vulnerability. (1) As the recommendation of solidity documentation [6], using the *Checks-Effects-Interactions* pattern [95], which delays interaction with external contracts after checking for conditions and effects to the state variables, can make the reentrancy vulnerability free. For example, in Figure 1(d), we can move the effect (line 4) to the front of the interaction (line 5) so that the balance is deduced before the transfer of Ether. Note that if the transfer fails, the effects of the transaction will be reversed, so that the balance is not changed. (2) As shown in the secure coding practices for the smart contract [81], we can add a *mutex modifier* to the function to avoid reentrancy vulnerability. Just like the protection of critical resources in the operating system, the mutex can guarantee that other calls to the function cannot be executed successfully until the current calling to the function is executed completely. Hence, the malicious external contract cannot transfer Ether again before its balance is deducted.

(V) SWC-115: Authorization through Tx-origin Vulnerability (TXO)

Vulnerability: A smart contract suffers from authorization through tx-origin vulnerability when misusing `tx.origin` for transaction authorization. In `Solidity`, the global variable `tx.origin` refers to the address of the EOA that initiates the transaction, while the global variable `msg.sender` refers to the address of the EOA or contract account that invokes the function. For example, if an EOA initiates a transaction by calling the function f_A of the contract A and the function f_A has an external call to the function f_B of the contract B, the values of `tx.origin` in function f_A and f_B are both the address of the EOA, while the values of `msg.sender` in function f_A and f_B are the address of the EOA and the address of the contract A, respectively. Figure 1(e) shows an example of TXO, which can be exploited in a way similar to a phishing attack. Specifically, the attacker can develop a phishing contract C_p, whose fallback function has an external call to function `sendTo` with the parameter set by the attacker. If the account of the `owner` address sends a transaction to the contract C_p, the fallback function of C_p will be invoked and the attacker will steal the money because the authorization at line 2 is passed.

Preventative Strategy: The most common way to avoid the tx-origin vulnerability is to replace the variable `tx.origin` with `msg.sender` for authorization. For example, in Figure 1(e), if the variable `tx.origin` at line 2 is replaced by `msg.sender`, the code at line 3 for transfer cannot be executed unless the `owner` address initiates a transaction that directly invokes the function `sendTo`.

These five types of vulnerabilities outnumber most other types and are of high impact [32, 89, 94]. Furthermore, the **Decentralized Application Security Project** [73] (**DASP**), which aims to discover smart contract vulnerabilities in the security community, covers the top 10 smart contract vulnerabilities ranked by the amount of financial loss in real-world projects. Among them, the top 4 vulnerabilities are: reentrancy (REN), access control that contains USI and TXO, arithmetic issues also known as IOU, and unchecked return values for low level calls (UCR) caused by smart contract programming or `Solidity` language and toolchain as inferred in Reference [20]. Also pinpointed by the survey [20], other vulnerabilities in the top 10 (not addressed by this article) are caused by the design and implementation of Ethereum, which cannot be repaired with automatic program repair techniques. Therefore, we focus on fixing these five vulnerabilities as described above.

2.3 Problem Definition

Our problem is defined as follows. Given a smart contract C, construct a smart contract C' such that C' satisfies the following.

- Soundness: C' is free of any of the above vulnerabilities.
- Correctness: For any legitimate transaction Tr, if Tr invokes the function F in C successfully, there exists a function F' in C' that can be invoked by Tr successfully and output the same result as F.
- Efficiency: C' execution speed and gas consumption are minimally different from those of C.

2.4 Summary of Existing Vulnerability Datasets

We introduce the existing public vulnerability datasets of the smart contracts of Ethereum. These datasets can be used as a benchmark to evaluate the effectiveness of existing vulnerability detection and repair tools.

2.4.1 Common Vulnerabilities and Exposures (CVE). **Common vulnerabilities and exposures** (**CVE**), which is sponsored by the US **Department of Homeland Security** (**DHS**) **Cybersecurity and Infrastructure Security Agency** (**CISA**), aims to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. A total of 531 vulnerability records for Ethereum smart contracts can be retrieved using keywords `smart contract` on the CVE website [27] until January 1, 2022. Additionally, the other 5 records [4] are also included, although the keywords are not explicit. Overall, the contract source code files are downloaded and classified according to the five types of vulnerabilities described in Section 2.2. After removing duplicates, we obtain 100 unique contracts.

2.4.2 Smart Contract Weakness Classification Registry (SWCRegistry). As mentioned in Section 2.2, SWCRegistry [72] offers a complete and up-to-date catalog of known smart contract vulnerabilities and anti-patterns of smart contracts along with real-world examples. A total of 37 vulnerability categories and 90 unique test cases (vulnerability samples) of smart contracts are in the SWCRegistry.

2.4.3 SmartBugs Curated Dataset ($SB^{CURATED}$). $SB^{CURATED}$ [93] is part of the executional framework SmartBugs [39]. It is organized according to the DASP taxonomy and contains 143 annotated vulnerable contracts with 208 tagged vulnerabilities. Note that $SB^{CURATED}$ was created from GitHub repositories, Blog posts, and the Ethereum network. Therefore, it partially overlaps with CVE and SWCRegistry.

The number of vulnerable functions in three datasets (CVE, SWCRegistry, and $SB^{CURATED}$) is shown in Table 2. Note that the total number is the result after removing duplicate contract source

Table 2. The Number of Vulnerable Functions for Five Vulnerability Types in Three Publicly Available Vulnerability Datasets

Datasets	IOU	UCR	USI	REN	TXO	Total
CVE	100	0	3	0	0	103
SWC	9	1	3	2	1	16
$SB^{CURATED}$	20	67	2	31	2	122
Total without duplication	121/228 (53%)	67/228 (29%)	7/228 (3%)	31/228 (14%)	2/228 (1%)	228

code files as [89]. Among them, the number of functions with IOU (121), UCR (67), and REN (31) reaches a reasonable amount to contain a variety of vulnerability patterns, while the number of functions with USI (7) and TXO (2) is much lower than other vulnerability types because the patterns of these two types are not diverse.

In addition, the SolidiFI [45] benchmark repository contains a dataset of buggy contracts injected by bug snippets, but there are three problems with synthetic bugs as stated in Reference [89]: (1) These vulnerabilities are independent of each other and have no interaction with the remaining internal functions of the original contract. (2) Their code logic is relatively simple, which involves only a few simple and commonly seen vulnerability patterns. And (3) Most vulnerabilities cannot be exploited in practice because the inserted buggy code is independent of the original business logic. Therefore, the SolidiFI benchmark repository is not adopted in this article.

3 MOTIVATION

In this section, we first discuss the limitations of existing smart contract vulnerability repair tools in Section 3.1. In particular, in Section 3.1.1, we provide a detailed account of the limitations of sGuard using a motivation example as shown in Figure 2. Additionally, we summary the limitation of SCRepair in Section 3.1.2. Then, we discuss the limitations of existing ML-based vulnerability detection works on smart contracts in Section 3.2.

3.1 Summary of Limitations of Existing Vulnerability Repairing Tools

Existing tools repair smart contracts at the source code level (e.g., sGuard [75], SCRepair [115]) or the bytecode level (e.g., EVMPATCH [90], SMARTSHIELD [117], ELYSIUM [101]). In this study, we focus on those at the source code level for two reasons: (1) The repair on the source code is conducive to the subsequent code audits that are an indispensable part for contract security and public trust. (2) This study is an extension and enhancement of the entire workflow of sGuard, which aims to guarantee that the contracts are vulnerability-free, so the repair of the bytecode is out of scope. The limitations of the two existing source-code-level vulnerability repair tools, sGuard and SCRepair, are introduced below. In general, factors that affect the effectiveness of repair include the effectiveness of vulnerability detection, the accuracy of localization, and the correctness of the repair strategies.

3.1.1 Limitations of sGuard. sGuard is a rule-based vulnerability repair tool, which has a built-in vulnerability detector. To identify potential vulnerabilities, sGuard performs static analysis on the finite set of symbolic execution traces collected from the smart contract. sGuard defines the vulnerability as a set of symbolic execution traces that suffer from the vulnerability, which can be called *vulnerable symbolic traces* for short. For example, if a symbolic trace executes an opcode `CALL` and subsequently executes an opcode `SSTORE` that depends on the opcode `CALL` in the same function, then it is a vulnerable symbolic trace that suffers from reentrancy vulnerability. Correspondingly, the vulnerability localization of sGuard utilizes the source mapping provided by the `Solidity` compilers, which maps opcodes to AST nodes. The repair rules of sGuard are

```
1  + contract sGuard {
2  +   bool internal locked_;
3  +   constructor() internal {
4  +     locked_ = false; }
5  +   modifier nonReentrant_() {
6  +     require(!locked_);
7  +     locked_ = true;
8  +     _;
9  +     locked_ = false; }
10 + }
11 - contract Reentrancy_bonus {
12 + contract Reentrancy_bonus is sGuard {
13 -   function withdrawReward(address recipient) public {
14 +   function withdrawReward(address recipient) nonReentrant_ public {
15        uint amountToWithdraw = rewardsForA[recipient];
16        rewardsForA[recipient] = 0;
17        (bool success, ) = recipient.call.value(amountToWithdraw)("");
18        require(success);}
19 -   function getFirstWithdrawalBonus(address recipient) public {
20 +   function getFirstWithdrawalBonus(address recipient) public nonReentrant_ {
21      require(!claimedBonus[recipient]);
22      rewardsForA[recipient] += 100;
23      withdrawReward(recipient);
24      claimedBonus[recipient] = true;}
25 }
```

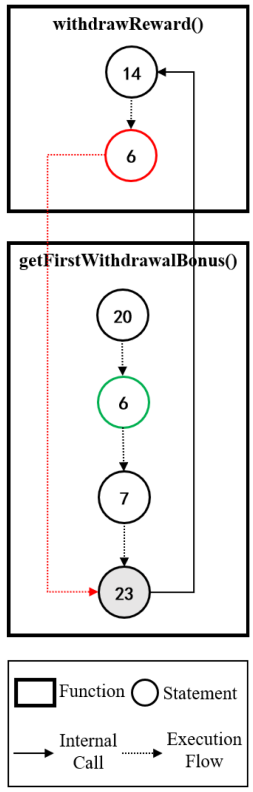

Fig. 2. A motivation example of a contract with a REN vulnerability repaired by sGuard from the dataset $SB^{CURATED}$.

designed to implement code transformation, which includes three basic operations: conditional insertion, variable substitution, and expression equivalence transformation.

As shown in Figure 2, its left side shows how a contract with the vulnerability REN is repaired by sGuard, which serves as a motivation example in this article.

In the original contract, as shown in the light red lines (-), the function `getFirstWithdrawalBonus`(line 19) has a REN vulnerability. To delve into details, it first checks if the claimed bonus of recipient is false (line 21); if it is, then gives the recipient 100 rewards (line 22), followed by a call to the function `withdrawReward` (line 23), which transfers the given rewards to the recipient (line 17), and finally sets the claimed bonus of recipient to true (line 24) to prevent claiming the reward again. However, before executing line 24, the transfer operation at line 17 may trigger a malicious recipient's callback function that calls the function `getFirstWithdrawalBonus`(line 19) again, leading to the illegal claim of the reward again.

Therefore, in the repaired contract, as shown in the light green (+) lines, sGuard adds the same mutex modifier `nonReentrant_` to both of these functions (lines 14 and 20) to avoid REN. The modifier adds the statements in lines 6 and 7 before the statements in lines 21 and 15, and adds the statement in line 9 after the statements in lines 24 and 18, which attempts to ensure that the function is not allowed to be called successfully until the previous invocation is executed completely.

However, the repaired contract results in the contract's original business logic being unable to execute successfully. As shown on the right side of Figure 2, we show an **interprocedural call graph** (**ICG**) that represents the internal calls between functions for the repaired contract, and utilize the line number of some key statements to show the execution flow. When calling the function `getFirstWithdrawalBonus` (line 20), the check in line 6 will pass because the variable `locked_` is initialized to false and then this variable is set to true in line 7. Next, the statement in line 23 calls the function `withdrawReward` (line 14), but the check in line 6 will fail, resulting in the entire invocation being a failure.

In the following, we illustrate three critical limitations of sGuard through this motivation example.

- **SG-LIM1.** In terms of vulnerability detection, sGuard produce some commonly seen false positives, which may result in repeated insertion of patches. Ignoring the vulnerability prevention code [79, 113] is the root cause of this limitation. For example, Figure 2 shows an example of sGuard that repairs REN by adding the `nonReentrant_` modifier to the vulnerable function `getFirstWithrawalBonus`. However, the repaired function still conforms to the REN detection rule of sGuard (a trace where the opcode `SSTORE` executes after the opcode `CALL`), which will be repeatedly repaired if it was input to sGuard.
- **SG-LIM2.** In terms of vulnerability localization, the vulnerability-free code that shares some opcodes with the vulnerable symbolic traces may be affected by the repair rules of sGuard, which may cause unexpected behaviors. The root cause of this limitation is that the vulnerable symbolic traces are symptoms of detected vulnerabilities, rather than the exact locations that should be modified according to the repair rules. For example, the original and vulnerable function `getFirstWithrawalBonus` (f_g) in Figure 2 has an internal call to the bug-free function `withdrawReward` (f_w) at line 23, which means f_g shares some opcodes with f_w. It shows a typical vulnerability of cross-function reentrancy as defined in sGuard [75], which can be exploited only by a call chain that contains at least two functions. In this case, the attacker can only withdraw money recursively by calling f_g, which is the only function that needs to be patched. However, sGuard inserts the same mutex modifier `nonReentrant_` into these two functions, which changes the original business logic of the contract. In detail, for the version patched by sGuard as shown in Figure 2, if a user calls f_g, the assignment statement (line 7) will first set the global variable `locked_` to `true`. Then, the function call statement (line 23) calls f_w that has the same modifier `nonReentrant_`. Next, the same conditional statement (line 6) will be executed in f_w, but it cannot be passed. Therefore, the transaction calling f_g will always be rolled back.
- **SG-LIM3.** In terms of repair strategies, the patch code inserted by sGuard may have weak protection, and the expression transformation may change the semantics of the original code if the expression has side effects. The root cause of this limitation is that some special cases are beyond the scope of the repair rules of sGuard. Additionally, there is only one repair strategy for each vulnerability type in sGuard. However, the tradeoff between more complicated repair rules and less gas consumption should be made if there are alternative repair strategies. For example, as shown in Figure 2, the visibility of the mutex variable `locked_` (line 2) is `internal`, which denotes the contract that inherits the contract `sGuard` which can change this mutex variable. It is reported to be vulnerable by CVE-2020-19765 [3], which may cause the access check for REN prevention ineffective. Moreover, the `Solidity` document [6] recommends the Conditions-Effects-Interaction mode to avoid REN, which can be considered as a better repair strategy of REN than mutex lock because it saves gas consumption. A more complicated repair rule is however required in such a case.

— **SG-LIM4.** In terms of efficiency, the time consumption of sGuard is unacceptably prolonged as the complexity of the contract increases. As shown in Reference [75], when the maximum time consumption per contract is 5 minutes, sGuard times out on 1,767 (35.34%) out of 5,000 contracts. The main reason is the path explosion problem of the vulnerability detection functionality of sGuard. Specifically, there are many contracts whose loops require a large number of unrollings.

3.1.2 Limitations of SCRepair. SCRepair [115] is a search-based and gas-aware vulnerability repair tool that adopts a genetic search technique to generate and select mutants that can be considered as candidate patches. SCRepair relies on existing vulnerability detection tools (Slither, Oyente) to detect and locate vulnerabilities, and these existing tools are also used as part of the criteria to evaluate the correctness and quality of the patch candidates. If all test cases pass and no vulnerabilities are detected by existing tools, the contract is considered repaired. However, according to the experimental results presented in SCRepair [115], the limitations of SCRepair are summarized as follows.

— **SCR-LIM1.** In terms of vulnerability detection and localization, Slither and Oyente are integrated by SCRepair to detect and locate the code that needs to be repaired. However, existing vulnerability detection tools have inconsistent results and a large number of false positives, as shown in many works [32, 45, 89]. The false positives and false negatives of existing tools and their inconsistencies affect the correctness of vulnerability repair.

— **SCR-LIM2.** In terms of repair rules, the goal of the mutation of SCRepair is only to pass test cases and ensure that no targeted vulnerability is found by one of the detection tools, so that the correctness of the generated patches must be verified manually. However, compared to fixed templates, the mutations of SCRepair may be incomprehensible, so more manual effort is required to verify the correctness.

— **SCR-LIM3.** In terms of efficiency, SCRepair consumes a lot of time and memory during the repair process. As shown in the experimental results of SCRepair [115], in their dataset that contains 20 contracts, 5 (25%) contracts are not repaired successfully due to out of memory error, and the average time consumption of 17 (68%) repaired contracts is 25 minutes. During the repair process of SCRepair, the execution time of the test case consumes the most computational resources, and the time budget of the genetic algorithm is at least one hour.

3.2 Summary of Limitations of Existing ML-Based Vulnerability Detection Works

Considering the limitations of existing vulnerability repair tools as mentioned above, we think machine learning technique can guide existing repair tools to be more advanced, especially with respect to performance. However, we find that although there are many existing ML-based vulnerability detection works on smart contracts, none of them have been used in existing vulnerability repair works. Hence, we follow the latest survey paper [97] to research existing ML-based vulnerability detection works as shown in Table 3. We summarize three inherent methodological limitations that hinder ML-based works from guiding repair works as follows.

— **ML-LIM1.** The labeling methods of existing works lack of manual confirmation process, which results in a large amount of benign code may be labeled as vulnerable, and is harmful to the supervised learning model. As shown in Table 3, except for the works [65, 121] and [33], the labeling technique for other existing works relies only on existing tools (e.g., Slither, Oyente) applying traditional approaches (e.g., static analysis, symbolic execution). However, many empirical studies [32, 89] show that these state-of-the-art tools have an incredible high false positive rate. Without human review of the results reported by the tool,

Table 3. Existing ML-Based Vulnerability Detection Works

Year	Work	Technique	Dataset Source	Training Set	Test Set	Labeling Method	Feature Source	Open Source
2018	[99]	DL	Ethereum	64%	20%	Maian	Opcode	Y
2019	SoliAudit [60]	ML	Etherscan	80%	20%	Oyente&Remix	Opcode	N
2019	[69]	ML	Etherscan	80%	20%	Mythril&Slither	Source Code	N
2020	Contractward [106]	ML	Etherscan	70%	30%	Oyente	Opcode	N
2020	[110]	DL	Etherscan	#N	#N	#N	Opcode	N
2020	[46]	DL	Etherscan	#N	#N	MAIAN	Opcode	Y
2020	[58]	DL	Etherscan	70%	30%	Mathx	Source Code	N
2020	TMP/DR-GCN [121]	DL	Etherscan	20%	80%	Manually	Source Code	Y
2021	AME [65]	DL	Etherscan	20%	80%	Manually	Source Code	Y
2021	Eth2Vec [11]	DL	Etherscan	90%	10%	#N	Opcode	Y
2021	[96]	DL	Etherscan	60%	70%	#N	Opcode	N
2021	[66]	DL	Ethereum	80%	20%	Oyente&Mythril&Dedaub	Opcode	N
2021	ESCORT [111]	DL	Etherscan	70%	30%	Slither&Ethainter	Source Code	N
2021	[33]	ML	Etherscan	#N	#N	Manually	Transaction	N
2021	[9]	ML	Etherscan	#N	#N	Etherscan	Transaction	N
2022	xFuzz [76]	ML	Etherscan	70%	30%	Solhint&Slither&Securify	Opcode&Source Code	N

[a]#N denotes that there is no explicit explanation in the article.

the labeling of the data set is unconvincing, and the model is bound to be misled into learning a large number of false positives. Although the works [121] and [65] label the source code of contracts in a purely manual way, the labeling method is rather crude (e.g, if the function possesses at least one invocation to `call.value`, it is labeled as potentially affected by the REN vulnerability.) In addition, the object of the manual labeling in the study [33] is transactions of contracts, which is not within the scope of this article.

— **ML-LIM2.** The program features as model inputs designed by existing works are insensitive to subtle differences between the vulnerable code and the repaired code. As shown in the eighth column of Table 3, more than half of the works extract merely opcode features, about a quarter extract merely source code features. However, as reported in Reference [114], continuous vectors extracted from the opcode or source code alone are not sufficient to train high-performance models. Hence, [114] enriches the vectors with additional static features of the source code. However, the work [114] aims to improve the performance of the fuzzing tool, so it achieves a high recall (0.95) but very low precision (0.26). Additionally, two works [33] and [9] extract features from transactions that are not program representations and are therefore outside the scope of this study.

— **ML-LIM3.** The training set and the test set in existing work are labeled by the rule-based tools. However, these tools tend to report fixed code patterns. Therefore, even if the model performs well on the test set, it may perform poorly on real-world vulnerable contracts. As shown in the fifth and sixth columns of Table 3, most of the studies divide the labeled dataset, which is labeled by one or more existing rule-based tools, into the training set and the test set using a certain proportion. Without evaluation based on a publicly available vulnerability dataset, the effectiveness of the model is unconvincing. In particular, if the model is not interpretable, it may be negatively affected by the distribution characteristics specific to a dataset, and thus learn rules unrelated to vulnerabilities [19].

4 APPROACH OVERVIEW

In this section, we present an overview of our approach as shown in Figure 3, which can be divided into two phases: the ML-based vulnerability detection phase in Section 4.1 and the rule-based vulnerability repair phase in Section 4.2. According to the limitations mentioned in Section 3, we present the key challenges of these two phases and the process to address the corresponding

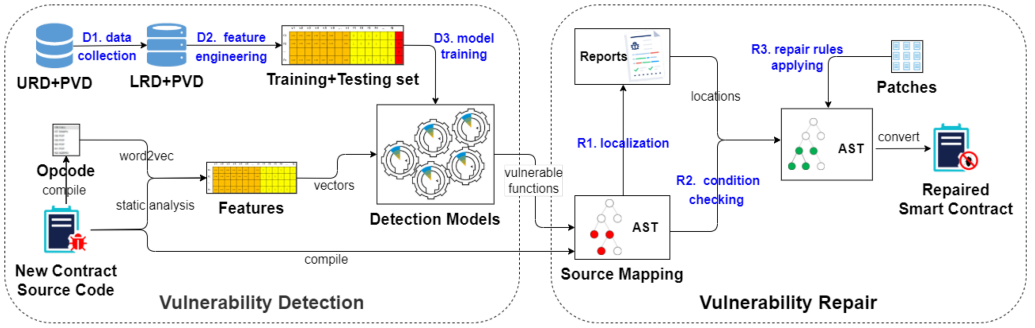

Fig. 3. Approach workflow.

limitations. Besides, we clarify the main differences and similarities between sGuard and sGuard+ in Section 4.3.

4.1 ML-Based Vulnerability Detection Phase

According to the limitations of sGuard (**SG-LIM1**, **SG-LIM4**) and SCRepair (**SCR-LIM1**) in Section 3.1, the key challenges in vulnerability detection are as follows.

(1) How to reduce false positives of existing rule-based vulnerability detection tools?
(2) How to improve efficiency while detecting as many vulnerabilities as possible?

To address these two challenges, sGuard+ utilizes the machine learning technique for vulnerability detection to enhance the effectiveness and efficiency. We start with a feature engineering exercise which considers both the features (Vul-Fs) that characterize the symptoms of vulnerabilities, and the features (Pre-Fs) that characterize the methods of vulnerability prevention. Vul-Fs are inspired by the detection rules of state-of-the-art vulnerability detection tools, and Pre-Fs are designed according to the defensive code of vulnerability, as shown in Refs. [20, 30, 72, 113]. Then, the training set is labeled on the basis of the results of the state-of-the-art tools. Since existing tools have inconsistent results, the results are manually confirmed to avoid false positives and false negatives. Unlike the training set, the test set is collected from publicly available vulnerability datasets (PVD) dataset, as shown in Section 2.4. In addition, PVD is augmented with the equivalent variants as done in [89] and the code obfuscations of BiAn [116] (see details as Section 5.2.1). The effectiveness of machine learning models in the test set represents the generalizability. Lastly, the tree-based machine learning model is selected for training because it has interpretability.

As shown in the left part of Figure 3, there are three main steps to get detection models: data collection, feature engineering, and model training.

First, for the step **D1.data collection**, the unlabeled real-world contract dataset (URD) provided by the empirical study [89] is an available standard dataset as part of a reliable benchmark suite. To build the training set, sGuard+ labels each function in URD based on the detection results of three state-of-the-art detection tools (two static tools Slither and Securify, and one symbolic execution tool Mythril). These three tools are used in previous work [29, 32, 89] for evaluation, which means that these tools are practically effective. Note that, in order to reduce the false positives and false negatives of three tools, the detection results are confirmed manually for solving the problems mentioned in the limitation of existing ML-based works (**ML-LIM1**). We collect existing publicly available vulnerability datasets shown in Section 2.4 as the test set, which aims to avoid the limitation of existing ML-based works (**ML-LIM3**).

Second, for the step **D2.feature engineering**, each function of a contract in LRD is characterized to a 39-dimensional vector by sGuard+. It contains 23-dimensional opcode features and

16-dimensional source code features. Specifically, the source code of a contract is compiled to get the opcode of each function, and then the opcode of each function is embedded by Word2Vec into a 20-dimensional vector. Furthermore, the opcode of each function is also used to identify 3-dimensional opcodes that are crucial for vulnerabilities to improve the vulnerability sensitivity. In addition, sGuard+ performs static analysis by Slither, which has the comprehensive built-in code analysis, to extract 16-dimensional source code features that contain Vul-Fs and Pre-Fs for each function for breaking though the limitation of existing ML-based works (**ML-LIM2**).

Third, for the step **D3.model training**, sGuard+ applies four typical tree-based binary classification models, namely **Decision Tree** (**DT**), **Random Forest** (**RF**), Adaptive Boosting (ABoost), and XGBT, to select the best performing model. Before training the models, sGuard+ adopts a commonly used oversampling algorithm, namely Borderline-SMOTE [50], to handle the imbalance in the number of vulnerable and non-vulnerable samples. Borderline-SMOTE improves the effectiveness of oversampling based on the SMOTE algorithm by only using the minority examples near the borderline to synthesize new samples. Finally, sGuard+ trains a binary classification model for each vulnerability type, because a function may have multiple vulnerabilities. A function will be fed to each of the five models for testing, and all reported vulnerabilities will be patched sequentially during the repair phase.

After getting the detection models, as shown at the bottom left of Figure 3, given a new smart contract source code, the vulnerability detection workflow of sGuard+ can be roughly divided into two steps to get the input and output of the models. First, to generate the input to the model, sGuard+ extracts the features of the new contract as in the step D2.feature engineering, and then gets the vectors of feature representation of the functions. Each vector is composed by a continuous distributed vector of opcode represented by Word2Vec, and a discrete distributed vector extracted by simple static analysis from the opcode and source code (see details in Section 5.2.2). Second, sGuard+ inputs the feature vectors into the models to detect vulnerable functions that should be repaired in the next phase.

4.2 Rule-Based Vulnerability Repair Phase

According to the limitations of sGuard (**SG-LIM2**, **SG-LIM3**) and SCRepair (**SCR-LIM2**, **SCR-LIM3**) for the localization and repair of vulnerabilities, the key challenges in this phase are twofold as follows.

(1) How to identify the particular lines of code that need to be repaired?
(2) How to improve the correctness of the repair rules?

To address the first challenge, sGuard+ applies an algorithm (see details in Algorithm 1) to localize the particular code lines that need to be patched according to the repair strategies. To address the second challenge, sGuard+ designs repair rules according to the official documentation (e.g., Solidity Documentation [6], SWCRegistry [72], Openzeppelin [79]) that provide secure remediation strategies and patterns, to guarantee that the patches are effective and bug-free. Furthermore, to preserve the original business logic of contracts, sGuard+ defines more precise conditions under which a repair rule can be safely applied and extends new repair rules for cases outside the scope. Note that patches with minimal code changes are preferred by sGuard+ to reduce gas overhead.

As shown in the right part of Figure 3, there are three main steps to repair vulnerabilities: localization, condition checking, and application of repair rules.

First, for the step **R1.localization**, the source mapping provided by `Solidity` compilers (e.g., solc) is utilized to find the line number of vulnerable code. sGuard+ localizes the code that needs to be patched at the function-level and the statement-level according to different patch methods. For function-level localization, the patch code can be inserted as a modifier of a vulnerable function.

For statement-level localization, sGuard+ traverses each statement in the function to identify the code that needs to be patched.

Second, for the step **R2.condition checking**, as mentioned in SG-LIM3, the repair rules of sGuard for REN and IOU are problematic in some cases. To preserve the original business logic of contracts, we set the conditions to check if the repair rules can be applied. For the repair rules of REN, the ICG is proposed to identify the calling relationship between functions. If there is more than one function with REN in the same call chain, sGuard+ will add mutex lock modifiers that depend on different mutex variables to the functions (see details in Figure 5 in Section 6.3). For the repair rules of IOU, sGuard+ checks whether the lvalue of an assignment expression has side effects and applies different repair rules accordingly (see details in Section 6.3).

Third, for the step **R3.repair rules applying**, the repair rules are designed to improve those of sGuard (see details in Figure 4 in Section 6.2). In general, sGuard+ not only refines the original rules of sGuard for REN and IOU by considering the precise conditions, but also extends the new repair rules to reduce gas consumption and repair two new types of vulnerability (UCR and USI). In general, there are three steps to repair a vulnerable contract. First, sGuard+ takes advantage of the source mapping to map the number of lines to the AST nodes. Then, the localized AST nodes are modified by the code transformation of repair rules. Finally, the modified AST is converted into a final source file of the repaired contract.

4.3 Comparison of sGuard and sGuard+

Both sGuard and sGuard+ are self-contained vulnerability repair tools with built-in vulnerability detection modules. sGuard employs a static analysis approach based on symbolic execution traces for vulnerability detection, while sGuard+ uses a machine learning approach based on feature engineering (see details in Section 5.2). Therefore, sGuard localizes vulnerabilities by directly mapping symbolic execution traces to source code, whereas sGuard+ designs a new algorithm to localize the source code that needs to be patched based on the vulnerable function signatures reported by the models (see details in Section 6.1). Both sGuard and sGuard+ perform vulnerability repairs using pre-designed repair rules.

In the vulnerability detection phase, sGuard defines which symbolic execution traces are vulnerable, which is adopted by sGuard+ as critical vulnerability features. For example, sGuard identifies a TXO vulnerability if a trace executes an opcode `CALL` that depends on an opcode `ORIGIN`. Consequently, we design two-dimensional features to indicate the presence of `CALL` and `ORIGIN`, respectively. However, to enrich the vulnerability features, we also refer to the detection rules of the state-of-the-art tools in feature engineering process. Although sGuard theoretically promises the soundness of static analysis, it exhibits poor practical effectiveness and performance due to the problem of path explosion. In contrast, sGuard+ outperforms sGuard in practice, thanks to its rich features, but it cannot guarantee theoretical soundness.

In the vulnerability repair phase, sGuard designs repair rules for REN, IOU and TXO, respectively. sGuard+ corrects the rules of sGuard to avoid possible changes to the original business logic and extends new rules to reduce gas overhead (see details in Section 6.2). Specifically, for REN, sGuard+ adapts the rule of sGuard according to the ICG of functions to guarantee the correct semantics, and designs a new rule that is applied preferentially because of its low gas overhead. For IOU, sGuard+ follows the rule of sGuard for arithmetic operators (e.g., +, -) but restricts the rule of sGuard for arithmetic assignment operators (e.g., +=, -=) on expression without sides effects to avoid incorrect repair, and then designs a new rule for expressions with side effects. For TXO, sGuard+ follows the rule of sGuard without modification. Additionally, sGuard+ adds new repair rules for UCR and USI vulnerabilities.

Table 4. The Detection Rules of Slither, Securify and Mythril

Tools	Mechanism	Vul	Rules
Slither	IR	UCR	$CALL(\_, \text{call()/send()}) \wedge Return(var\_x) \succ \neg\, Read(var\_x)$
		USI	$CALL(\_, \text{suicide()/selfdestruct()}) \wedge \neg\, IsProtected$
		REN	$CALL(\_, \text{call()}) \succ Write(var\_g)$
		TXO	$IsCondition \wedge Read(\text{tx.origin}) \wedge \neg\, Read(\text{msg.sender})$
Securify	DSL	UCR	$some\ \text{call}(L\_1, Y, \_, \_)\ all\ \text{goto}(L\_2, X, \_).\ MayFollow(L\_1, L\_2) \succ \neg\, MayDepOn(X, Y)$
		REN	$some\ \text{call}(L\_1, \_, \_, \_)\ some\ \text{sstore}(L\_2, \_, \_).\ MustFollow(L\_1, L\_2)$
Mythril	SE	IOU	$\exists I_0 \in \{SUB\}\ .\ P(I_0) \wedge (op_1 > op_0)$ or $\exists I_0 \in \{ADD, MUL\}\ .\ P(I_0) \wedge (op_1 + op_0 > 2^{23} - 1)$
		UCR	$\exists I_0 \in \mathbf{C}, I_1 \in \{STOP, RETURN\}, I_0 \succ I_1, \forall Retval(I_0)\ .\ P(I_1)$
		USI	$\exists I_0 \in \{SELFDESTRUCT\}, \forall addr\ .\ P(I_0) \wedge (tx.origin == addr)$
		REN	$\exists I_0 \in \mathbf{C}, I_1 \in \{SSTORE, SLOAD, CREATE, CREATE2\}, I_0 \succ I_1\ .\ P(I_1)$
		TXO	$\exists I_0 \in \{ORIGIN\}, I_1 \in \{JUMPI\}, I_0 \succ I_1\ .\ P(I_1)$

Overall, the main difference between sGuard and sGuard+ lies in their distinct approaches to vulnerability detection. Additionally, the repair rules of sGuard and sGuard+ are roughly the same, but sGuard+ corrects and extends the rules of sGuard to improve the correctness and performance.

5 VULNERABILITY DETECTION VIA MACHINE LEARNING

In this section, we elaborate on the vulnerability detection phase by machine learning. To begin with, the detection rules of three state-of-the-art tools (Slither, Securify, and Mythril) are introduced in Section 5.1, which represent multiple symptoms of vulnerabilities. Then, in Section 5.2, we introduce three steps of model training: data collection, feature extraction, and model selection. Furthermore, we evaluated the F1 scores of four tree-based models on the training and test sets to select the best model.

5.1 The Detection Rules of Existing Tools

The detection rules of Slither, Securify, and Mythril are formalized in Table 4. In the following, we present the details of their approach.

Slither. Slither works by converting contracts into an intermediate representation (IR), namely SlithIR. The built-in code analysis of Slither contains three parts [38]: ① Read/Write (R/W), which represents the read / write operator for the variables. ② **Protected Functions** (**PF**), which represent that the function is not the constructor and that the variable `msg.sender` is not directly used in a comparison. ③ **Data Dependency Analysis** (**DDA**), which contains the data dependency in the function and the date dependency in the entire contract. As shown in Table 4, *Read*(var) denotes reading variables; *Write*(var) denotes writing variables; var_g denotes global variables; var_s denotes state variables; var_x denotes a specific variable; *isProtected* denotes that the function is a PF; *CALL*(_, $func$) denotes calling the built-in function $func$ of `Solidity`; *Return*(var) denotes the call return value; *IsConditon* denotes conditional statements; $\succ$ denotes the execution order in the control flow. In general, UCR will be reported if the call return value is not read in the control flow after the call, because if the return value is not read then it must not be checked. USI will be reported if the function that calls the built-in function `selfdestruct` or `suicide` does not check the identity of the caller, `msg.sender`, because anyone may destruct the contract if there is no authorization mechanism for callers. REN will be reported if there is a call to the built-in function `call` followed by a write operation on a global variable, which is the most obvious requirement for REN. TXO will be reported if a conditional statement reads the variable `tx.origin` for comparison, but does not read the variable `msg.sender`, because `tx.origin` should

not be used for authorization unless it is checked to ensure that `tx.origin` and `msg.sender` are the same address.

Securify. The rules of Securify are specified in a domain-specific language (DSL). As shown in Table 4, *MayFollow*(L_1,L_2) denotes the instruction (opcode) at label L_2 may follow that at label L_1. *MustFollow*(L_1,L_2) denotes the instruction (opcode) at label L_2 must follow that at label L_1. *MayDepOn*(X,Y) denotes the value of Y that may depend on the tag T. instr(L,Y,$X1$, …,Xn) (e.g., call(L_1,Y,_,_)) denotes operations of the instructions. Specifically, instr is the name of the instruction, L is the label of the instructions, Y is the variable that stores the result of the instruction (if any), and $X1$, …, Xn are the variables given to the instruction as arguments (if any). The dot in the middle denotes the condition on the left that must hold for *some*/*all* instructions on the left. In general, Securify detects UCR by checking that the `CALL` instruction is not followed by a `GOTO` instruction which may depend on the return value. Securify detects REN by checking that there is a `CALL` instruction that must be followed by a write operator to the storage variables.

Mythril. Mythril is implemented based on the symbolic execution technique. As shown in Table 4, *P*(I) denotes the constraint on the path to instruction I, $op0$ and $op1$ denote the left and right variables of dyadic arithmetic operations (e.g., $op0 + op1$), **C** denotes the instruction set {`CALL`, `DELEGATECALL`, `STATICCALL`, `CALLCODE`}, *Retval*(I) denotes the return value (True or False) of an instruction I. In brief, IOU will be identified by Mythril if a constraint path with arithmetic overflow or underflow can be satisfied. UCR will be identified if a constraint path to an instruction in {`STOP`, `RETURN`} can be satisfied regardless of whether the return value of the instruction in **C** is true or false, which means that the return value is unchecked. USI will be identified if a constraint path to the instruction `SELFDESTRUC` can be satisfied with the address of any of the callers. REN will be identified if the stage changes after a message call. TXO will be identified if a control flow decision is influenced by the variable `tx.origin`.

In summary, despite the different approaches, these three tools rely on similar features to identify vulnerabilities. For example, the source code/opcode Vul-Fs of UCR, USI, and TXO are roughly summarized as: ① call the built-in functions `addr.call()` or `addr.send()`/ execute the opcode `CALL`, ② call the built-in functions `suicide()` or `selfdestruct()`/execute the opcode `SELFDESTRUCT`, ③ read the variable `tx.origin` as a condition/execute the opcode `ORIGIN`, respectively. Additionally, writing to the storage variable after calling the function `addr.call()` or executing the opcode `CALL` (*write-after-call*) is a necessary condition for REN. However, *write-after-call* is not a sufficient condition for REN [113]. There is a Pre-F that uses the mutex lock modifier, which can prevent REN without failing the condition of *write-after-call*. Therefore, to reduce false positives and false negatives of existing tools, sGuard+ extracts features by integrating key Vul-Fs and Pre-Fs.

5.2 Model Training for Vulnerability Detection

In this section, we discuss data collection in Section 5.2.1 and introduce feature engineering in Section 5.2.2, followed by candidate model selection in Section 5.2.3.

5.2.1 Data Collection.

Data Source. Smart-Contract-Benchmark-Suites [89] is a representative dataset containing three typical categories: unlabeled real-world contracts (UR), contracts with manually injected bugs (MI) and confirmed vulnerable contracts (CV). Among them, UR provides 45,622 unique unlabeled real-world contracts with source code, which are collected from verified contracts on Etherscan [5] by searching for all smart contract addresses provided by BigQuery [47]. MI provides labeled contracts from SolidiFI, and CV provides vulnerable contracts collected from the CVE and SWCRegistry datasets.

Table 5. The Number and the Percentage of Vulnerable Functions in the Publicly Available Vulnerability Dataset (PVD)

IOU	UCR	USI	REN	TXO	Total
121/247 (49%)	**67**/247 (27%)	**10**/247 (4%)	**31**/247 (13%)	**18**/247 (7%)	247

Table 6. The Number of Vulnerable Functions in the Labeled Real-World Contracts Dataset (LRD)

Tool	IOU	UCR	USI	REN	TXO
Slither (Sl.)	-	280/1,025	190/190	382/81,037	194/194
Securify (Se.)	-	379/28,807	28 /28 (V2)	373/13,232	10 /10 (V2)
Mythril (My.)	13,825	222/522	285/285	381/40,069	343/3,226
Manually Confirmed Results	7,782	825	54	69	434

[a]**V2** denotes that the vulnerability type is only supported by the version 2.0 of Securify, which only detects contracts with the Solidity vision greater than or equal to 0.5.8.
[b]The symbol **/** denotes the minimum sample size with 95% confidence level and 5% confidence interval on the left, and the number of vulnerabilities reported by the tools on the right.

Overall, we build the **publicly available dataset** (**PVD**) as test set, which contains the vulnerable contracts in the labeled datasets of CVE, SWCRegistry and SmartBugs. Furthermore, considering that the number of USI (7) and TXO (2) as shown in Table 2 is insufficient for evaluation, we apply the variant approaches of Reference [89] and the obfuscation of the code by BiAn [116] to amplify the number and diversity of these two types of vulnerability. In total, the number of vulnerable functions in PVD is shown in Table 5. In particular, the test set is updated until January 1, 2022 (the CVE is continuously updated), and duplicate contrasts are removed as in Reference [89].

Besides, we label UR as follows to build the **labeled real-world contracts dataset** (**LRD**) as training set. The contracts in the test set are all excluded from the training set.

Data Labeling. In LRD, each function is labeled based on the results of three state-of-the-art vulnerability detection tools, namely Slither, Securify, and Mythril. The main reason for choosing these three tools is twofold. First, the empirical study [32] shows that these three tools perform best in the five types of vulnerability. Second, these three tools are the top 3 smart contract audit tools according to the report [49] of a company with the level of an international **managed security service provider** (**MSSP**).

Each function is annotated as vulnerable or non-vulnerable. We do not simply label functions based on votes from two or more tools as in Reference [114] because we find that there may be one tool that outperforms the others for a certain vulnerability type. For example, Mythril reports far more TXO vulnerabilities than the other two tools, but it also has the most false positives.

Therefore, in order to guarantee the accuracy of labeling, the results of three tools are manually confirmed by the three authors of this article, each of whom has more than 3 years of code auditing experience for smart contracts. The number of detection results of the three tools on UR and the manually confirmed results are shown in Table 6.

However, we find that many vulnerable functions reported by a particular tool are clones because the detection rules of a tool are fixed and unique. Although we obtain reported vulnerabilities from three tools instead of one to ensure the diversity of vulnerabilities, manual confirmation is then faced with such a large number of reported results that it is impossible to confirm completely. Therefore, we first randomly sample the vulnerabilities reported by each tool, and then manually verify the samples of each tool to obtain as diverse vulnerability code as possible. Specifically, as shown in Table 6, the results of each tool for each vulnerability type except IOU, which contains more than 500 samples, are sampled for manual confirmation to obtain a confidence level 95% and

a confidence interval 5% on whether the sample is representative of all results. According to the results in Table 6, the minimum sample size we need to confirm is 280, 379, 222 and 382, 373, 381 for the UCR and REN results of Slither, Securify and Mythril, respectively, and 343 for the TXO results of Mythril.

In addition, Mythril reports arithmetic overflow vulnerability based on symbolic execution and constraint solving. Since the static features of IOU are less accurate than symbolic execution or dynamic features, and the IOU are more common than other vulnerability types, we keep all IOU results of Mythril in order to preserve as many IOU patterns as possible. However, we find that if a function f_n without IOU calls a built-in hash function (e.g., `keccak256()`) or an internal function with IOU, then the function f_n will be redundantly reported as vulnerable. Therefore, we remove redundant functions, and the remaining results are labeled as IOU by default, although it may not directly result in a loss of funds.

In general, a function is labeled as vulnerable if it is manually confirmed on the detection results of any tool. The function that is not reported by any of the three tools is labeled non-vulnerable.

Data Processing. We extract the feature vector for each function in the dataset using the method described in Section 5.2.2, and obtain a total of 761,415 feature vectors for 45,622 contracts. To avoid bias caused by data duplication (due to many function clones existing in smart contracts [54]), we remove all repeated feature vectors and finally leave 135,283 feature vectors. Among them, only about 2.8% (3,818/135,283) feature vectors are labeled vulnerable. However, the extreme imbalance of vulnerable and non-vulnerable samples will introduce an undesirable bias in the model and limit its predictive performance. Therefore, we use an oversampling technique based on the Borderline-SMOTE algorithm [50] to balance the ratio of vulnerable and non-vulnerable samples in the training set. Borderline-SMOTE is one of the most popular algorithms for oversampling based on improvements to the SMOTE algorithm.

5.2.2 Feature Engineering. In order to convert the code into compact and uniform length of the feature vector while retaining critical information of the vulnerabilities, we extract 39-dimensional features from each function by Slither. It contains 23-dimensional opcode features and 16 source code features.

Opcode Features: In EVM, each opcode is encoded as one byte, which means up to 256 unique opcodes. In fact, EVM has 77 valid opcodes if `PUSH1 PUSH32` are regarded as `PUSH`, `DUP1 DUP16` as `DUP`, `SWAP1 SWAP16` as `SWAP` and `LOG0 LOG4` as `LOG`, as shown in Table 1. Besides, there are some unassigned byte (e.g., 0x21 0x2f, etc.), called the INVALID opcode, which is not included because it is not actually implemented in the Ethereum client. We first use `Word2Vec` to represent the opcode of each function as a 20-dimensional vector. Then, we extract three binary features indicating if the opcodes `CALL`, `ORIGIN` and `SELFDESTRUCT` are executed in the function or not, respectively, which represent the properties strongly associated with the vulnerabilities. For example, as clarified in Section 5.1, the detection rules of Securify and Mythril for UCR and REN all start by identifying the presence of the opcode `CALL`. Furthermore, USI and TXO may be reported by Mythril if and only if the code executes the opcode `SELFDESTRUCT` and `ORIGIN`, respectively. Hence, compared with continuous features produced by `Word2Vec`, discrete binary features are beneficial for better partitioning feature space and improving the accuracy and efficiency of machine learning models.

Source code features: Most of the semantic information in the source code will be lost after the smart contract is compiled into bytecode, which leads to a high false positive rate. Therefore, the source code of each function is leveraged to extract 16 features, as shown in Table 7. Each source code feature represents a binary attribute with boolean values. Features are categorized

Table 7. Details of 16 Source Code Features

Name	Type	Description	Nature
unprotected_ren	bool	*IsCondition*∧*Read(X_g)* ≻ ¬*Write(X_g)* ≻ *CALL(_,call())* ≻ *Write(Y_g)*	Negation of Pre-F for REN
has_arithmetic_operator	bool	has arithmetic operators	Vul-F for IOU
dangerous_txorigin	bool	*IsCondition*∧*Read(X)*∧*DataDep(X,tx.origin)* ∧ ¬ *DataDep(tx.origin,msg.sender)*	Vul-F for TXO
unchecked_return_value	bool	*CALL(_,call()/send())* ≻ *Return(_,X)* ≻ ¬(*IsCondition*∧ *Read(X)*)	Vul-F for UCR
unprotected_suicide	bool	¬ *IsCondition* ≻ *CALL(_,suicide()/selfdestruct())*	Vul-F for IOU
has_high_level_call	bool	has an external call to a function of a contract at a certain address	Vul-F for REN and UCR
has_low_level_call	bool	has a call to the function `call()/delegatecall()/codecall()`	Vul-F for UCR
has_call	bool	has a call to the function `call()`	Vul-F for REN
has_transfer	bool	has a call to the function `transfer()`	Pre-F for REN
has_send	bool	has a call to the function `send()`	Pre-F for REN & Vul-F for UCR
has_internal_call	bool	has an internal call to a function in the same contract	Pre-F for IOU
is_visible	bool	the visibility of the function is public or external	Vul-F for REN and USI
has_condition	bool	has conditional statements	Pre-F for USI
has_modifier	bool	has custom modifiers	Pre-F for REN and USI
has_msg.sender	bool	read msg.sender variable	Pre-F for TXO
has_msg.value	bool	read msg.value variable	Vul-F for IOU

into two groups based on their correlation with vulnerabilities: (1) A feature that may lead to a vulnerability when its value is true is referred to as a vulnerability feature (Vul-F), as the symptoms of the vulnerabilities described in Section 2.2. (2) A feature that may defend against a vulnerability when its value is true is referred to as a prevention feature (Pre-F), as the preventative strategies described in Section 2.2. Existing work focuses on generalizing patterns related to Vul-Fs, while overlooking the role of the Pre-Fs. For instance, as shown in our motivation example in Figure 2, all three existing tools (Slither, Securify and Mythril) report that the repaired contract still contains a REN vulnerability, because the claimed bonus of the recipient is still updated to true (line 24) after transfer (line 17), which satisfies the *write-after-call* rule as summarized in Section 5.1. However, all of them overlook the protective effect of the mutex modifier. Accordingly, we not only enrich the Vul-Fs, but also focus on enriching the Pre-Fs to improve the accuracy of the model, particularly reducing the model's false positives.

We introduce the details of these features in Table 7 as follows. Note that, *DepData*(var_1,var_2) denotes that the variable var_1 has a data dependency on var_2, and the other symbols have the same meaning as described above in the detection rules of Slither. ① `unprotected_ren`, which is positively correlated with vulnerabilities, is designed to contain two parts, one (*CALL(_,call())* ≻ *Write(Y_g)*) is a Vul-F about write-after-call, and the other (*IsCondition*∧*Read(X_g)* ≻ ¬*Write(X_g)*) is the opposite of the Pre-F about mutex lock. For example, as in the example shown in Figure 2, the modifier `nonReentrant_` prevents REN by reading the global variable `locked_` to check that the condition is satisfied (line 6) and writing the variable to change its value (line 7) before executing the call to the function `addr.call()` (line 23). ② `has_arithmetic_operator` indicates that the function has an arithmetic operator that may cause IOU. Using safeMath functions [79] instead of arithmetic operators is recommended, since `Solidity` with the version lower than 0.5.0 does not check for overflow by default. Therefore, sGuard+ works to replace every arithmetic operator with the corresponding safeMath function. ③ `dangerous_txorigin` is designed to identify the conditional statement that reads the variable `tx.origin` without depending on the variable

`msg.sender`. This feature can prevent false positives that commonly exist in existing tools in the case `require(tx.origin==msg.sender)`. ④ `unchecked_return_value` is designed to identify no conditional statement to read the call return value. ⑤ `unprotected_suicide` is designed to identify no conditional statement before calling the `suicide` or `selfdestruct` function. Furthermore, for the remaining features, `has_high_level_call`, `has_low_level_call`, `has_call`, `has_transfer`, `has_send` and `has_internal_call` represents different call types in `Solidity`. In addition, the visibility of a function, conditional statements, custom modifiers, and read to the transaction-related variables (e.g., `msg.value` and `msg.sender`) are all significant semantic information for vulnerabilities.

5.2.3 Model Selection.

Model Selection. We choose the tree-based classification models because they have good interpretability compared with deep learning models [19]. This interpretability is crucial for this work for two main reasons: (1) We can trust the rationale behind the vulnerability detection model because the decision-making process (e.g., the paths in a decision tree) of the model can be explicitly observed. This aids in guiding the subsequent localization and repair phases, ensuring the correctness of the approach. (2) The interpretability of models is utilized in the numerous iterations of trial-and-error for feature engineering to analyze important features and discover more useful ones to improve the accuracy of models. In comparative experiments, four major tree-based binary classification models, namely DT, RF, Adaptive Boosting (AdaBoost), and XGBT, are evaluated as candidate models. DT is a supervised learning algorithm that recursively partitions the feature space of the training set to provide an informative and robust hierarchical classification model [71]. RF, AdaBoost, and XGBT belong to the family of ensemble methods that combine multiple base learners to create more accurate and robust models. Among them, RF builds multiple DTs and aggregates their predictions to make the final decision to improve accuracy and generalization[16]. AdaBoost combines multiple weak learners (typically DTs) to create a strong learner, which achieves high accuracy [88]. XGBT is an efficient and scalable implementation of Gradient Boosting similar to AdaBoost [24], which is known for its speed and performance. Note that, to improve the efficiency of tuning hyperparameters, we utilize random search [15] to obtain the optimal hyperparameter combination.

Metrics. There are four outcomes that could occur by models: (1) **True positives** (**TP**) mean the vulnerable functions are reported vulnerable correctly; (2) **True negatives** (**TN**) mean the functions without vulnerabilities are reported non-vulnerable correctly; (3) **False positives** (**FP**) mean the functions without vulnerabilities are reported vulnerable incorrectly; (4) **False negatives** (**FN**) mean the vulnerability functions are reported non-vulnerable incorrectly.

The three main metrics used to evaluate the performance of a model are Precision, Recall, and F1, as shown in Equation (1). Precision indicates the fraction of TPs among all vulnerable cases reported by a model. Recall indicates the fraction of TPs among all of the labeled vulnerability cases. F1 is a comprehensive evaluation metric that controls the same importance between precision and recall.

$$Precision = \frac{TP}{TP + FP}, \qquad Recall = \frac{TP}{TP + FN}, \qquad F1 = \frac{2 * Precision * Recall}{Presion + Recall} \tag{1}$$

Results Analysis. Table 8 shows the average precision, recall, and F1 scores of the four models in 10-fold cross-validation on the training set (LRD). The 10-fold cross-validation is repeated 10 times to reduce the randomness caused by partitioning the dataset. For each vulnerability type, XGBT outperforms the other three models for USI and REN, since its F1 score is the highest, while the F1 of XGBT is 0.01 (0.95–0.94) lower than the best AdaBoost model for TXO. The F1 of XGBT is the same as RF and AdaBoost for IOU (0.36) and the same as RF for UCR (0.95), which shows small

Table 8. The Precision, Recall and F1 Measure of Four Models for 10-fold Cross-Validation on the Training Set (LRD)

VUL	Model	Precision	Recall	F1
IOU	DT	0.19	0.42	0.26
	RF	0.32	0.44	**0.36**
	AdaBoost	0.36	0.38	**0.36**
	XGBT	0.29	0.46	**0.36**
UCR	DT	0.91	0.98	0.94
	RF	0.91	0.99	**0.95**
	AdaBoost	0.91	0.99	0.94
	XGBT	0.91	0.99	**0.95**
USI	DT	0.72	0.85	0.77
	RF	0.77	0.90	0.82
	AdaBoost	0.70	0.80	0.74
	XGBT	0.80	0.90	**0.84**
REN	DT	0.58	0.75	0.63
	RF	0.55	0.90	0.63
	AdaBoost	0.65	0.65	0.63
	XGBT	0.83	0.90	**0.82**
TXO	DT	0.88	1.00	0.93
	RF	0.90	1.00	0.94
	AdaBoost	0.91	0.99	**0.95**
	XGBT	0.90	0.99	0.94
Average	DT	0.66	0.80	0.71
	RF	0.69	0.85	0.74
	AdaBoost	0.71	0.76	0.72
	XGBT	0.75	0.85	**0.78**

Table 9. The Precision, Recall and F1 Measure of Four Models on the Test Set (PVD)

VUL	Model	Precision	Recall	F1
IOU	DT	0.49	0.24	0.32
	RF	0.51	0.22	0.31
	AdaBoost	0.25	0.9	0.39
	XGBT	0.51	0.72	**0.60**
UCR	DT	1.00	0.96	0.98
	RF	1.00	0.96	0.98
	AdaBoost	1.00	0.96	0.98
	XGBT	1.00	0.96	**0.98**
USI	DT	1.00	0.70	0.82
	RF	1.00	0.70	0.82
	AdaBoost	1.00	0.70	0.82
	XGBT	1.00	0.70	**0.82**
REN	DT	1.00	0.9	0.95
	RF	1.00	0.87	0.93
	AdaBoost	1.00	0.9	0.95
	XGBT	1.00	0.9	**0.95**
TXO	DT	1.00	0.94	0.97
	RF	1.00	0.94	0.97
	AdaBoost	1.00	0.56	0.71
	XGBT	1.00	0.94	**0.97**
Average	DT	0.90	0.75	0.81
	RF	0.90	0.74	0.80
	AdaBoost	0.85	0.80	0.77
	XGBT	0.90	0.84	**0.86**

differences between these models. Overall, the XGBT classifier model has the highest F1 score (0.78) because it has the best precision (0.75) and recall (0.85). In contrast, even though the recall (0.85) of RF is the best as XGBT, the precision (0.69) of RF is lower than XGBT, and the precision (0.66) of DT and the recall (0.76) of AdaBoost are the worst.

Table 9 shows the precision, recall, and F1 score of four models in the test set (PVD), which evaluates the generalization of four models on real vulnerabilities. The F1 (0.60) of XGBT is the best for IOU, although its F1 score is the same as that of RF and AdaBoost on the training set. It shows that XGBT has the best generalizability. The difference in the F1 score between XGBT and the other three models is slight for the remaining four types of vulnerability. The main reason is that the samples of these four vulnerability types in the test set are simpler than those of the IOU. In general, the F1 (0.86) of XGBT is the best, and the precision (0.90) and the recall (0.84) are also the best.

Considering that both precision and recall have a significant impact on the effectiveness of vulnerability repair, such as high recall can guarantee that more vulnerabilities are repaired and high precision can avoid affecting other non-vulnerable functions, we evaluate models mainly based on the F1 measure. Therefore, XGBT is selected to detect vulnerabilities in sGuard+ because the F1 of XGBT is the best in both the training set and the test set.

6 AUTOMATED VULNERABILITY LOCALIZATION AND REPAIR

In this section, we present the details of the vulnerability localization in Section 6.1 and the repair rules in Section 6.2. In addition, the case study of repair rules is introduced in Section 6.3 for a detailed explanation.

ALGORITHM 1: Vulnerability Localization

Input: Vulnerable Functions F
Output: Mapping From Function To Location M

foreach f *in* F **do**
  if $f.vul$ *is UCR* **then**
    foreach s_i *in* $f.S$ **do**
      r = GetLowLevelCallReturnValue(s_i,($Send, Call$))
      c := False
      foreach s_j *in AfterStatements*($f.S$,s_i) **do**
        if *CheckValue*(s_j, r) **then**
          c := True
      if *not* c **then**
        $M(f) \leftarrow s_i.line$
  if $f.vul$ *is REN* **then**
    foreach s_i *in* $f.S$ **do**
      if *HasLowlevelCall*(s_i,$Call$) **then**
        $l_c \leftarrow s_i.line$
        V := storage variables read in condition statements before s_i
        foreach s_j *in AfterStatements*($f.S$,s_i) **do**
          $v_w \leftarrow$ WriteStorageVariable(s_j,V)
          if v_w != $\emptyset$ **then**
            V_b := variables read in statements between s_i and s_j
            if v_w *not in* V_b **then**
              $l_w \leftarrow s_j.line$
        $M(f) \leftarrow (l_c, l_w)$
      $M(f) \leftarrow f.lines$
  if $f.vul$ *is TXO* **then**
    foreach s_i *in* $f.S$ **do**
      V := GetVariableReadInCondition(s_i)
      if `tx.origin` *in* V *and not DataDependency(* `msg.sender`,$V - \{tx.origin\}$ *)* **then**
        $M(f) \leftarrow s_i.line$
  if $f.vul$ *is IOU or USI* **then**
    $M(f) \leftarrow f.lines$
return M

6.1 Localization

Source mapping generated by the compiler of `Solidity` provides a mapping from the source code to the nodes in the AST or bytecode. Since sGuard leverages symbolic traces, which are generated based on a **control flow graph** (**CFG**) that is converted from bytecode, to detect vulnerabilities, sGuard uses the mapping from vulnerable opcode in the symbolic trace to the location of the source code. However, using machine learning models for vulnerability detection poses a challenge for localization, because models only report the signature of the vulnerable function and the type of vulnerability, but not the critical vulnerability statement in the function. To address this, we design an algorithm to locate the specific number of vulnerable statements. The details are presented

in Algorithm 1, which takes a set of instances of vulnerable functions F as input and returns a mapping from the vulnerable function to the corresponding line number of vulnerable statements. Each f in F contains the type of vulnerability and a set of statement instances S that are abstracted by Slither based on the control flow nodes [92].

We leverage static analysis to identify statements in vulnerable functions that need to be fixed by applying repair rules. To start with, we check the vulnerability type of the vulnerable function f. Next, we search the location M that needs to be patched for each vulnerability type as follows.

UCR (Lines 2 to 10): We first search the statement s_i that calls the function `addr.send()` or `addr.call()` and get the corresponding return value r by the function `GetLowLevelCallReturnValue` (lien 4). Then, we get the set of statements after s_i by the function `AfterStatements` and iterate through each statement s_j in this set (line 6). Each s_j will be judged by the function `CheckValue` whether the return value r is checked (line 7). If the variable r is not checked in the next statements, the line number of s_i is the place to be modified (line 10).

For instance, as shown in Figure 1(b), the statement at line 4 calls the function `call`, while the statements after line 4 do not check the return value of the function `call`. Therefore, line 4 is the location that should be patched by a conditional statement for checking the return value.

REN (Lines 11 to 23): We first search the statement s_i that calls the function `addr.call()` by the function `HasLowLe- velCall` (line 13), then the set of storage variables V that are read in condition statements before s_i will be collected (line 15). The set of statements after s_i collected by the function `AfterStatements` is iterated as s_j (line 16) for checking the write operator to storage variables that are stored on the blockchain. We search for the same storage variables v_w written by s_i and s_j by the function `WriteStorageVariable` (line 17). If v_w is not empty, we will check whether the variables in v_w are read by the statements between s_i and s_j in the same branch of the program (lines 19 to 20). If not, the line number of s_i and s_j (l_c, l_w) is the place to be modified (line 22). In addition, the range of line numbers of the function indicates the place where the modifier can be added (line 23).

For example, the REN example shown in Figure 1(d) displays a typical situation in which a condition statement on line 2 reads a storage variable `credit[msg.sender]`, and the statement (line 4) writes the variable `credit[msg.sender]` after the statement calling `addr.call()` (line 3). In this case, moving the statement at line 4 to the front of the statement at line 3 will satisfy the Conditions-Effects-Interaction mode. REN in this case can also be defensed by locking as shown in modifier `nonReentrant_`.

TXO (Lines 24 to 28): We first collect the variables V read by the conditional statement s_i by the function `GetVaria- bleReadInCondition` (line 26). If the variable `tx.origin` in V, we then check if the remaining variables except `tx.origin` in V have data dependencies on `msg.sender` by the function `DataDependency` (line 27). If not, the line number of s_i is the place to be modified (line 28).

For example, Figure 1(e) shows a TXO example. The condition statement at line 2 reads variable `tx.origin` and `owner` that does not depend on `msg.sender` in the function `sendTo`. Replacing `tx.origin` with `msg.sender` will defend this TXO.

IOU (Lines 29 to 30): For a vulnerable function with IOU, we locate the range of line numbers of the function, which contains all unsafe expressions that will be converted by the safe math functions.

For instance, as shown in Figure 1(a), if the function `sell` is reported to have IOU vulnerability, the range of line numbers for the entire function (lines 1 to 5) will be located and then all arithmetic expressions within the range, such as the subtraction on line 3 and the multiplication at line 4, will be patched by the safe math functions.

$$\frac{F \in \text{REN} \quad F \rightarrow F' \quad F'' = \text{lock}(F')}{F \rightarrow F''} \quad \text{(ChREN-1)}$$

$$\frac{S_1 \in \textbf{call} \quad S_2 \in \text{writeStmt} \quad S_1 \rightarrow S_1' \quad S_2 \rightarrow S_2'}{S_1; \ldots; S_2 \rightarrow S_2'; S_1'; \ldots} \quad \text{(ChREN-2)}$$

$$\frac{E_1 \rightarrow E_1' \quad E_2 \rightarrow E_2' \quad E_3 \rightarrow E_3' \quad \text{opFunc} \in \text{safeFunc}(\odot)}{E_1 = E_2 \odot E_3 \rightarrow E_1' = \text{opFunc}\,(E_2', E_3')} \quad \text{(ChIOU-1)}$$

$$\frac{E_1 \rightarrow E_1' \quad E_2 \rightarrow E_2' \quad \text{opFunc} \in \text{safeFunc}(\oplus) \quad E_1 \notin \text{sideEffect}}{E_1 \oplus E_2 \rightarrow E_1' = \text{opFunc}\,(E_1', E_2')} \quad \text{(ChIOU-2)}$$

$$\frac{E_1 \rightarrow E_1' \quad E_2 \rightarrow E_2' \quad \text{opFunc} \in \text{safeFunc}(\oplus) \quad E_1 \in \text{sideEffect} \quad a \in Symbol}{a[E_1] \oplus E_2 \rightarrow tmp = E_1'; a[tmp] = \text{opFunc}\,(a[tmp], E_2')} \quad \text{(ChIOU-3)}$$

$$\frac{E \rightarrow E'}{E \rightarrow E'\texttt{[tx.origin/msg.sender]}} \quad \text{(ChTXO)}$$

$$\frac{E \in \textbf{call} \quad E \rightarrow E'}{E; \rightarrow \text{success} = E; \text{require}(\text{success});} \quad \text{(ChUCR)}$$

$$\frac{F \in \text{USI} \quad F \rightarrow F' \quad F'' = \text{check}(F')}{F \rightarrow F''} \quad \text{(ChUSI)}$$

$\rightarrow$ denotes syntax exchange; $\oplus$ denotes arithmetic assignment operators (e.g. +=, -=); $\odot$ denotes arithmetic operators (e.g. +); ... denotes a sequence of program code statements; / denotes the replacement of the preceding variable with the following variable; **safeFunc** denotes input a arithmetic operator to get the corresponding safemath function; **sideEffect** denotes a set of program code with side effect; **call** denotes a set of program code for transaction by Call function; **send** denotes a set of program code for transaction by Send function; **writeStmt** denotes a set of program code with write operations; **REN** denotes a set of functions with REN vulnerability; **USI** denotes a set of functions with USI vulnerability; **lock** is a map which takes in a function and gets a function with modifier prevented REN vulnerability.

Fig. 4. Repair rules of sGuard+. (Cyan represents the improved rules based on sGuard. Blue represents the newly expanded rules).

USI (Lines 29 to 30): For a vulnerable function with USI, we locate the range of line numbers of function, which indicates the place where to add the permission check modifier that ensure only the contract creator is allowed to destroy the contract.

For instance, as shown in Figure 1(c), if the function `sendTo` is reported to have USI vulnerability, the range of line numbers for the entire function (lines 1 to 4) will be located, and then the function will be patched by adding a permission check modifier before the keyword `public` at line 1.

Overall, since the statement is abstracted based on the control flow nodes, which contains the variables reading and dependency information that can search directly, the time consumption of the algorithm is mainly two layers traversal of the function statements. In general, the complexity of the vulnerability localization algorithm is $O(n^2)$.

6.2 Repair Rules

The repair rules of sGuard+ are shown in Figure 4. sGuard+ addresses the limitations of the repair rules of sGuard by considering the interprocedural calls for REN and the side effects of the arithmetic expression for IOU. Note that the repair rules for new vulnerability types (UCR and USI) and the gas-saving repair rule for REN are newly introduced by sGuard+. In the following, we present how the rules in Figure 4 are designed.

ChREN-1: For any function f in a smart contract, if there is a function f' with REN, add the mutex lock to f', and then we get the repaired function f''.

ChREN-2: For any function f with REN, if there are two statements s_1 and s_2 in f, which satisfy the condition that s_1 calls the built-in function `call.value()` *and the immediate next statement s_2 writes to the storage variable, switch the order of s_1 and s_2.*

As shown in Reference [75], sGuard supports the repair rules on REN (no matter intra-function and cross-function), TXO and IOU. Among them, the repair rule of REN as shown in ChREN-1 is to add a modifier `nonReentrant_`, which is a mutual exclusion mechanism by a mutex storage variable `lock_`, which can prevent the recursive call before the first call finishes from having any effect. However, as we mentioned in Section 3.1.1, if function *A* has an internal call to function B which uses the same mutex, the call to function A will always be reverted. Therefore, as shown on the right side of Figure 2, we design the ICG constructing the internal call relationships between functions to prevent functions in the same internal call chain from relying on the same mutex.

Moreover, compared with the modifier mutex, the Condition-Effects-Interaction mode [6] means constructing a nature mutual exclusion in business logic without increasing the amount of code. For instance, as shown in the simple REN example in Figure 1(d), the statements on lines 2, 3 and 4 represent the conditions, interaction and effects, respectively. The Conditions-Effects-Interaction can be conformed if we move the statement on line 4 before the statement on line 3, and we can find that the storage variable `credit[msg.sender]` is a natural mutex to prevent REN. Therefore, to fix REN with minimal cost, the repair rule ChREN-2 is designed to prevent REN by moving the statement S_2 with Effect to the front of the statement S_1 with Interaction.

ChIOU-1: For any function f with IOU, if there are expression e_1, e_2 and e_3 in f, which form an arithmetic statement s : $e_1 = e_2 \odot e_3$; (e.g., `a = b + c;`) potentially causing integer overflow or underflow, convert s to a new statement s' : $e_1' = opFunc(e_2', e_3')$ (e.g., `a = safeADD(b,c);`).

ChIOU-2: For any function f with IOU, if there are expressions e_1 and e_2 in f, which form an arithmetic statement s : $e_1 \oplus e_2$; (e.g., `a += b;`) that causes integer overflow or underflow, convert s to a new statement s' : $e_1' = opFunc(e_1', e_3')$ (e.g., `a = safeADD(a,b);`) if and only if e_1 has no side effects (e.g., e_1 is not `arr[a+=1]`).

ChIOU-3: For any function f with IOU, if there are expression e_1 with side effects and e_2 in f, which forms an arithmetic statement s : $a[e_1] \oplus e_2$; (e.g., `arr[a+=1] += b;`) causing an integer overflow or underflow, add a new statement $tmp = e_1'$ (e.g., `tmp = a+=1;`) that assigns e_1 to a new temporary variable and then convert s to a new statement s' : $a[e_1] = opFunc(a[tmp], e_3')$ (e.g., `a[temp] = safeADD(a[tmp],b);`).

The repair rules of IOU (ChIOU-1 and ChIOU-2) are the same as those of sGuard, which replaces arithmetic operators with safemath functions that revert the transaction if IOU occurs. ChIOU-1 and ChIOU-2 show that the arithmetic operators (e.g., +, -, *, /, etc.) and the compound arithmetic assignment operators (e.g., + =, - +, * =, /=, etc.) are replaced with the corresponding safemath functions. However, the original semantics will be changed if the replacement is performed directly on a compound arithmetic assignment operator that contains side effects in an lvalue. Hence, we design ChIOU-3 to address the side effect of expressions on semantics by introducing the temporary variable `tmp`.

ChTXO: For any function f with IOU, if there is a conditional expression e (e.g., `require(tx.origin==owner)`) reading the variable `tx.origin` in f, replace `tx.origin` with `msg.sender`, to obtain the repaired expression e'.

For TXO, `tx.origin` denotes the address of the original initiator of the full call chain of transactions, which can be exploited by phishing attacks by using the caller's address as the authentication of its unknown calling node. In contrast, `msg.sender` denotes the address of the current caller, which means that there is no intermediate call node that may deceive the caller. ChTXO shows the same repair rule for TXO as sGuard, which replaces `tx.origin` of vulnerable expression with `msg.sender`.

ChUCR: For any function f with UCR, if there is an expression e (e.g., `msg.sender.call.value("1 ether")`*) in f, which invokes a low-level call method but its return value is unchecked, convert the statement* `e;` *to a new statement s_1* `:success = e;` *assigning the return value to the variable* `success`*, and then add a new conditional statement s_2* `:require(success);` *immediately following s_1.*

The UCR repair rules check the return value of the expression of the function call by adding the built-in error handling function `require()` in `Solidity` as shown in ChUCR.

ChUSI: For any function f in a smart contract, if there is a function f' with USI, add a permission check (e.g., `require(msg.sender == owner)`*) on f', and then we get the repaired function f''.*

The repair rule of USI ChUSI is similar to that of ChREN-1, which adds the authentication modifier to vulnerable functions.

6.3 Case Study

In this section, we show the application of repair rules in some specific examples to clarify the rationality and correctness. The purpose of the case study is mainly to show the improvement of sGuard+ over sGuard (the rules of ChREN-1, ChREN-2, ChIOU-2, and ChIOU-3), so the correct repair rules (the rules of ChIOU-1 and ChTXO) of sGuard are not redundantly demonstrated through examples. In general, the patch code provided by the rule-based repair method is fixed, which means that the correctness of the repair depends on the correctness of the patch code and the rationality of the repair rules. However, after manually auditing the contracts repaired by sGuard on PVD, we find a weak prevention patch code as shown in Figure 5 for REN in implementation, and two special cases as shown in Figures 5 and 7 that are beyond the scope of the repair rules for REN and IOU, respectively. Therefore, in order to improve the correctness of the repair rules of sGuard, we amend the weak patch code and improve the rationality of the repair rules with conditional constraints. Furthermore, we adopt a variety of repair strategies as shown in Figure 6 based on research about state-of-the-art recommended remediation and security patches for each considered vulnerability to reduce repair costs and cover more complex situations.

Repair rule ChREN-1. To show the application of ChREN-1, we slightly modify the example shown in Figure 2 to move the statement in line 14 after line 16 so that the function `withdrawReward` is also REN. Figure 5 shows the example of ChREN-1 applied in a vulnerability case in which a REN function `getFirstWithdrawalBonus` has an internal-call to another REN function `withdrawReward` in the same contract `Reentrancy_bonus`. In this case, adding the same modifier `nonReentrant_` for two REN functions by sGuard is incorrect for the same reason as explained in Section 3.1.1.

Therefore, according to ChREN-1, we first construct the ICG of functions in this contract. Then we check the relationship of all REN functions on the ICG. The different modifier, such as `nonReentrant_1` and `nonReentrant_2`, will be added to the REN functions if there is a linear internal-call between them and the same modifier will be added if there is not inter-procedural call between them. Note that we only report an alarm if there is a ring of inter-procedural calls between REN functions because it cannot be repaired unless the program logic is adjusted. Moreover, in the implementation, we change the visibility of variables treated as mutexes such as `lock_1` and `lock_2` from `internal` to `private` to avoid introducing new vulnerabilities.

Repair rule ChREN-2. For REN, the optimal repair method is to satisfy the Conditions-Effects-Interaction mode instead of using a mutex [30] which has potential dangers such as deadlock. Hence, we design the repair rule ChREN-2 to identify the conditions that satisfy the mode and perform code transformations. Figure 6 shows an example of ChREN-2 applied in Figure 1(d).

For Figure 1(d), the statement `if` in line 2 is a condition, the transaction statement in line 3 is an interaction, and the arithmetic assignment statement in line 4 is an effect. According to ChREN-2, first, the interaction in line 3 is S_1, and we will find the effective statement S_2 in line 4 which writes a storage variable `credit[msg.sender]` read by conditional statement in the same conditional branch

```
+ contract sGuardPlus {
+   bool private locked_1;
+   bool private locked_2;
+   constructor () internal {
+     locked_1 = false;
+     locked_2 = false;
+   }
+   modifier nonReentrant_1() {
+     require(!locked_1);
+     locked_1 = true;
+     _;
+     locked_1 = false;
+   }
+   modifier nonReentrant_2() {
+     require(!locked_2);
+     locked_2 = true;
+     _;
+     locked_2 = false;
+   }
+ }
- contract Reentrancy_bonus {
+ contract Reentrancy_bonus is sGuardPlus {
-   function withdrawReward(address recipient) public {
+   function withdrawReward(address recipient) nonReentrant_2 public {
      uint amountToWithdraw = rewardsForA[recipient];
      (bool success, ) = recipient.call.value(amountToWithdraw)("");
      require(success);
      rewardsForA[recipient] = 0;
    }
-   function getFirstWithdrawalBonus(address recipient) public {
+   function getFirstWithdrawalBonus(address recipient) nonReentrant_1 public {
      require(!claimedBonus[recipient]);
      rewardsForA[recipient] += 100;
      withdrawReward(recipient);
      claimedBonus[recipient] = true;
    }
}
```

Fig. 5. An example of ChREN-1 applied in Figure 2. Note that the statement in red font is added to make the function `withdrawReward` has a REN.

```
  function withdraw(uint amount) public {
    if (credit[msg.sender]>= amount) {
-     require(msg.sender.call.value(amount)());
-     credit[msg.sender] -= amount;
+     credit[msg.sender] -= amount;
+     require(msg.sender.call.value(amount)());
  } }
```

Fig. 6. An example of ChREN-2 applied in Figure 1(d).

```
+ contract sGuard {
+   function add_uint8(uint8 a, uint8 b) internal pure returns (uint8) {
+     uint8 c = a + b;
+     assert(c >= a);
+     return c;
+ }  }
- contract IOUEample {
+ contract IOUEample is sGuard {
    uint8[] public arr = [1,2,3];
    uint8 idx = 0;
    function iou_vul(uint8 amount) public{
      ...
-     arr[idx += 1] += amount; // If amount is 10, idx will be 1, and arr will be [1, 12, 3].
+     arr[idx = add_uint8(idx, 1)] = add_uint8(arr[idx = add_uint8(idx, 1)], amount);
+     // If amount is 10, idx will be 2, and arr will be [1, 2, 12]. This is inconsistent with the original result.
      ...
  } }
```

(a) The incorrect repair by sGuard for IOU due to side effects of the arithmetic expression.

```
  function iou_vul(uint8 amount) public{
    ...
-   arr[idx+=1] += amount;
+   uint tmp = idx += 1;
+   arr[tmp] = add_uint8(arr[tmp], amount); // If amount is 10, idx will be 1, arr will be [1, 12, 3].
    ...
  }
```

Fig. 7. An example of ChIOU-3.

with S_2. Then, we will check whether the variables read or write by the statements between S_1 and S_2 contain `credit[msg.sender]`. Finally, if it does not contain, we will move the statement that is an effect before the statement that is an interaction, as shown in Figure 6.

Repair rule ChIOU-3. The conversion of the IOU repair rules provided by sGuard will change the semantics of the code if the left expression of an arithmetic assignment statement has side effects. As shown in Figure 7(a), the statement on line 13 will be repaired by sGuard to the statement on line 14, which is slightly different from the original semantic because an extra 1 is added to the variable `idx`.

Therefore, ChIOU-2 requires the left expression of the statement without side effects and ChIOU-3 is designed to avoid repeated arithmetic assignment operations by introducing a temporary variable `tmp`. As shown in Figure 7(b), first, we assign the expression `idx += 1` with side effects to a temporary variable `tmp`. The statement `tmp = idx += 1` is added. Then, the original expression `idx += 1` is replaced by `tmp` and the original statement is converted to `arr[tmp] += 1;`. Finally, the repair rule ChIOU-2 is applied in the statements `tmp = idx += 1` and `arr[tmp] += 1;` as shown in lines 4 and 5, respectively.

7 EVALUATION

To evaluate the effectiveness and efficiency of sGuard+, we construct extensive experiments to answer the following **research questions** (**RQs**):

RQ1: How *effective* is sGuard+, when compared with state-of-the-art vulnerability repair tools?

— **RQ1.1:** Does the repaired contract change the original business logic of the contract?
— **RQ1.2:** What is the vulnerability repair capability of sGuard+?

RQ2: How *effective* is each key step of sGuard+, namely vulnerability detection, localization and repair?

— **RQ2.1:** How accurate is the XGBT model of sGuard+, when compared with state-of-the-art tools?
— **RQ2.2:** Do the XGBT model of sGuard+ reduce false positives, and are the Pre-Fs helpful?
— **RQ2.3:** Can the localization algorithm find where the true positive vulnerability should be fixed?
— **RQ2.4:** Do the repair rules of sGuard+ ensure that the transformed code is syntactically correct?

RQ3: How *efficient* is sGuard+ and how much gas overhead of the patches is introduced by sGuard+?

— **RQ3.1:** What is the time and memory performance of sGuard+?
— **RQ3.2:** Does sGuard+ reduce the gas overhead compared with sGuard?

We first compare the repair capability and correctness of sGuard+, sGuard and SCRepair in RQ1 by reproducing the vulnerabilities exploiting transactions and historical transactions for regression testing on repaired contracts. Then, since the repair correctness of sGuard+ is affected by three key steps: (1) ML-based vulnerability detection; (2) vulnerability localization; and (3) code transformation to apply repair rules, we separately evaluate the effectiveness of each step in RQ2. Finally, in RQ3, the efficiency of three tools (SCRepair, sGuard and sGuard+) is evaluated to illustrate the computing resource consumption of repair and the gas overhead introduced by patches.

7.1 Experiment Setup

Baseline Tools. In evaluation, sGuard is a main baseline tool because sGuard+ is an improvement over sGuard in our study. Furthermore, as a source-code-level repair tool, SCRepair is also included for comparison with sGuard+. Note that the source code of SCRepair cannot be executed successfully (also mentioned in article [101]) due to the lack of critical dependency components, and we have to reuse their experimental results for comparison. The bytecode-level repair tools are excluded because they are inevitably more prone to alter the original code semantics due to the high repair difficulty, resulting in poor correctness compared with source-code-level tools. To substantiate this claim, we evaluate the correctness of ELYSIUM, which outperforms existing bytecode-level tools, by conducting the experiment related to RQ1.1. Subsequently, we find that 92% of the historical transactions failed after the repair, and even some transactions that cannot be reproduced locally before repairing are executed successfully (see open-sourced details in Reference [8]). However, undeniably, bytecode-level repair holds significant importance for contracts without available source code.

Besides, in order to evaluate the effectiveness of the XGBT of sGuard+ individually, we select three state-of-the-art nonML-based tools (Mythril, Slither and Securify) and two state-of-the-art ML-based tools (TMP/DR-GCN and Eth2Vec) as baseline tools. First, the advanced nature of these three non-ML-based tools is obvious as mentioned in Section 5.2.1, so they are used not only for data labeling but also as baseline tools. Second, as shown in the summary of existing ML-based works in Table 3 of Section 3.2, there are five open-source works, namely [46, 99], TMP/DR-GCN [121], AME [65], and Eth2Vec [12]. However, the granularity of vulnerability detection in [99] and [46] is a contract rather than a function, and the work [99] does not classify a specific vulnerability type. Additionally, the works [46] and AME [65] do not open-source the code for data pre-processing [77]. We try to reproduce the code according to the article [65], but we find that some of the super-parameters of the model (e.g., dimensions of model inputs and outputs) are not provided in the article, making it impossible for us to reconstruct the model architecture for

Table 10. Datasets for Tool Evaluation

Dataset	Labeled	Smart Contracts	Vulnerable Functions	Research Questions
PVD	Yes	234	247	All
SCRD	No	17	#	All except RQ2.2
PVD-R	Yes	234	0	RQ2.2

[a]# denotes that the vulnerable functions are not publicly confirmed waiting to be detected by tools.

feature generation. Therefore, only the work TMP/DR-GCN [121] and Eth2Vec [12] can be used as a baseline tool. Note that TMP/DR-GCN [121] proposes two models: TMP and DR-GCN. Although TMP outperforms DR-GCN in Reference [121], we evaluate both of them in the experiment.

Datasets for Tool Evaluation. As shown in Table 10, we use three datasets to evaluate sGuard+ comparing with the state-of-the-art tools: (1) The publicly available vulnerability dataset (PVD), as mentioned in Section 5.2.1, which contains 234 unique contract source code files that are labeled with 247 vulnerable functions. (2) The dataset of SCRepair (SCRD), which contains 17 contracts from real-world projects. And (3) The repaired PVD (PVD-R), which contains the repaired contracts that are manually modified and confirmed according to the public remediation cases provided by SWC.

These three datasets communicate different perspectives. First, the vulnerabilities in PVD are confirmed by official organizations [27, 105], which represents the ground truth for each vulnerability type. Hence, PVD is an essential benchmark for evaluating not only the effectiveness of vulnerability detection but also the correctness of vulnerability repair. Second, to compare with SCRepair, we evaluate the effectiveness of sGuard+ in SCRD. Different from PVD, the vulnerabilities in SCRD are unconfirmed, so it is necessary to rely on existing tools for detecting vulnerabilities and performing manual confirmation. Last, to answer the RQ2.2, which contains the evaluation of the ability of tools to recognize the vulnerability prevention code, we manually construct PVD-R because the number and location of vulnerabilities as well as the corresponding remediation are explicit on PVD.

Experimental Environment. sGuard+ is implemented based on Python v3.8.2 and Node.js v16.14.2. The compiler of smart contract is required to be installed locally. All experiments are conducted on an Ubuntu 18.04 LTS machine equipped with an Intel(R) Xeon(R) Gold 6226 CPU @ 2.70 GHz and 187 GB of memory.

7.2 RQ1: How Effective is sGuard+, When Compared with State-of-the-Art Vulnerability Repair Tools?

To answer the RQ1, we evaluate the effectiveness of sGuard+ by comparing it with sGuard and SCRepair in PVD and SCRD. To begin with, we verify the repair correctness of sGuard+ and sGuard on PVD and SCRD by reproducing the vulnerability exploiting transactions and historical transactions. The repair correctness of SCRepair is not verified using the same approach because SCRepair cannot be executed so that the information of the repaired contracts is unavailable. Then, we show the repair results of sGuard+ and sGuard on PVD for each type of vulnerability, which can truly and accurately evaluate the effectiveness of tools because PVD represents the ground truth of vulnerabilities. Finally, we show the repair results of sGuard+, sGuard, and SCRepair on SCRD for each vulnerability type, which can evaluate the effectiveness of tools on real-world and complicated contracts in SCRD. Note that we reuse the original experiment results of SCRepair on SRCD from the study in Reference [115].

Table 11. Transactions for Regression Testing (RT)

Tool	Repaired Contracts (PVD and SCRD)	Historical Transactions (Top 350)	Successfully Reproduced Transactions On The Local Net	Failed Transactions In RT	
				T_e	T_b
sGuard	64	7,982	4,206 (53%)	1	420
sGuard+	176	16,755	9,109 (54%)	156	0

[a]Note that sGuard and sGuard+ fix 3 and 5 kinds of vulnerabilities, respectively.

RQ1.1: Does the repaired contract change the original business logic of the contract?

Correctness Validation. Given a transaction T_e exploiting a vulnerability V_i of a vulnerable contract C_v, and a set of benign history transactions T_b of C_v, V_i is considered to be repaired correctly if the repaired contract C_r can fail T_e and pass all T_b.

Failing the T_e means that the original vulnerable functions cannot be exploited, and passing all the T_b implies that the original business logic of contracts is not broken. In our evaluation, we first exploit vulnerabilities by a sequence of deliberately crafted transactions. Then we reproduce these transactions on the repaired contracts to check whether these vulnerabilities are successfully defended. Moreover, we perform regression testing (RT) to check whether the original business logic is normal. The available historical transactions are collected for each contract, and then they are reproduced on the original and repaired contracts, respectively, to check whether they will be executed successfully as before the repair.

We use the Truffle framework [25] to deploy contracts, and claw the top 350 historical transactions for each contract as test cases for RT from Etherscan so that the huge number of similar transactions can be restricted to reduce unnecessary effort. However, due to the difference between the Ethereum network and the local network (e.g., the block number, timestamps, and the addresses of external contracts called by transactions), not all historical transactions can be reproduced successfully, which is also mentioned in [75, 101, 115]. As shown in Table 11, the historical transactions of sGuard are different from those of sGuard+ because sGuard only repairs three types of vulnerabilities in PVD and SCRD. The number of contracts repaired by each tool is shown in the second column. Moreover, the third column shows the number of historical transactions and the fourth column shows the number of these transactions reproduced successfully on the original contracts. We use successfully reproduced transactions as test cases on repaired contracts for RT.

As shown in the last column of Table 11, there are a total of 421 (10%) transactions that failed to execute in contracts repaired by sGuard and 156 (2%) transactions that failed to execute in contracts repaired by sGuard+. Among them, we distinguish two types of transactions: T_e (exploitation of vulnerabilities transaction) and T_b (benign history transaction). In terms of vulnerability prevention, contracts repaired by sGuard correctly defend against 1 T_e with IOU, and contracts repaired by sGuard+ correctly defend against 141 and 15 T_e with UCR and IOU, respectively. In terms of impact on the original business logic of the contract, there are 420 T_b failures to execute in contracts repaired by sGuard, while contracts repaired by sGuard+ pass all T_b. Among these 420 T_b failed to execute, 408 T_b failed because there are 6 contracts that fail to compile due to unreliable code transformation implementation (see details in RQ2), 11 T_b failed due to out-of-gas, and 1 T_b failed because the patch code changes the original business logic as LIM3 of sGuard in Section 3.1.1. Overall, sGuard+ defends against 155 (156-1) more T_e than sGuard without changing the original business logic.

RQ1.2: What is the vulnerability repair capability of sGuard+?

Results on PVD. Table 12 shows the correct repair results for each vulnerability type on PVD by sGuard and sGuard+. A repair is considered correct when the patched function is one of the

Table 12. The Correct Repair Results for Each Vulnerability Type on PVD

Dataset	Vulnerability Types	sGuard Correctly Repair / Ground Truth	sGuard+ Correctly Repair / Ground Truth
PVD	IOU	25/121 (21%)	85/121 (70%)
	UCR	-	64/67 (96%)
	USI	-	7/10 (70%)
	REN	9/31 (29%)	28/31 (90%)
	TXO	17/18 (94%)	17/18 (94%)
Total		51/170 (30%)	199/247 (81%)

[a]- denotes the vulnerability type unsupported by the tool.
[b]Since SCRepair cannot be executed successfully, its results on PVD are not available.

Table 13. The Correct Repair Results for Each Vulnerability Type on SCRD

Dataset	Vulnerability Types	SCRepair Correctly Repair/Detect	sGuard Correctly Repair/Detect	sGuard+ Correctly Repair/Detect	Total
SCRD	IOU	0/12	4/5	13/13	15/24
	UCR	18/28	-	30/30	30/30
	USI	-	-	0/0	0/0
	REN	1/1	1/1	0/0	1/1
	TXO	-	0/0	0/0	0/0
Total		19/41	5/6	43/43	46/55

[a]- denotes the vulnerability type unsupported by the tool.

labeled vulnerable functions on PVD, and it is confirmed by correctness validation. Each row of Table 12 represents a vulnerability type. Each cell represents the number of vulnerabilities that are repaired correctly by the tool and the number of vulnerabilities marked on PVD. Compared with sGuard, the repair rate of IOU and REN of sGuard+ increased by 49 pp (from 21% to 70%) and 61 pp (from 29% to 90%), respectively. Furthermore, sGuard+ supports more types of vulnerabilities than sGuard such as UCR and USI. Overall, the vulnerability repair rate of sGuard+ on PVD increased by 51 pp (from 30% to 81%) over sGuard.

Results on SCRD. Table 13 shows the correct repair results for each contract on SCRD by SCRepair, sGuard and sGuard+. Each cell represents the number of vulnerable functions and how many of them are correctly repaired by the tool for each vulnerability type. Since SRCD contains 17 contracts that are not publicly confirmed, the ground truth of vulnerabilities in these contracts is not available, which is different from PVD. Nevertheless, we count the number of vulnerabilities for each vulnerability type detected correctly by each tool, and manually confirm the correctness of the repair.

In detail, the three tools have not discovered any USI and TXO vulnerabilities. For IOU, there are 12 vulnerable functions reported by Oyente, while none of them are correctly repaired by SCRepair according their experiment results in Reference [115]. sGuard reports 5 IOUs, 4 of which are correctly repaired. sGuard+ detects 13 vulnerable functions and repairs all of them correctly. For UCR, SCRepair and sGuard+ correctly repair 18 and 30 vulnerable functions, respectively. For REN, although SCRepair shows that it correctly repaired 3 REN in its experiment results, we find that 2 of them are FPs. SCRepair and sGuard correctly detect and repair 1 REN missed by sGuard+. As shown in Figure 8, the main reason why the REN of the function `issue` can be exploited is that the protection of the modifier `onlyOwner` is invalid, which is also known as Taint for Owner Vulnerability [55]. Hence, it is ignored by sGuard+ because it has a defensive feature about the modifier. In general, the vulnerability repair rate of sGuard+ on

```
contract Ownable {
    ...
    function Ownable() {
        owner = msg.sender;}
    modifier onlyOwner() {
        require(msg.sender == owner);
        _;}
    ...
}
contract Issuer is Ownable {
    ...
    function Issuer(address _owner, address _allower, StandardToken _token) {
        ...
        token = _token;}
    function issue(address benefactor, uint amount) onlyOwner {
        if(issued[benefactor]) throw;
        token.transferFrom(allower, benefactor, amount);
        issued[benefactor] = true;
        issuedCount += amount;}
}
```

Fig. 8. The Banana Coin Solidity code at 0xD113244B9049943D4bc6fEf3048d24EDf92dd788, which is an FN of REN by sGuard+.

SCRD increased by 44 pp (from 19/55 to 43/55) over SCRepair and 69 pp (from 5/55 to 43/55) over sGuard.

Answer to RQ1: sGuard+ performs best in evaluating the effectiveness among vulnerability repair tools. Overall, the vulnerability repair rate of sGuard+ increased by 51 pp over sGuard on PVD, and 44 pp over SCRepair on SCRD. Additionally, sGuard+ has no impact on the original business logic in the regression testing, while sGuard fails 420 benign test cases.

7.3 RQ2: How Effective is Each Key Step of sGuard+, Namely Vulnerability Detection, Localization, and Repair?

To answer the RQ2, we first evaluate the precision, recall and F1 score of the machine learning models (XGBT) of sGuard+ in PVD by comparing them with four nonML-based state-of-the-art vulnerability detection tools (Slither, Securify, Mythril and sGuard) and two ML-based tools (TMP/DR-GCN and Eth2Vec). Second, to evaluate the effectiveness in identifying the vulnerability prevention code, we manually construct the repaired PVD (PVD-R) that contains the repaired contracts that are manually confirmed according to the public remediation cases provided by SWC. The FPs of existing tools, which are caused by ignoring vulnerability prevention code, can be revealed by PVD-R. Then, to further evaluate the effectiveness of localization that determines where to repair, we count the number of incorrect patch locations for sGuard and sGuard+ based on the vulnerable functions reported correctly. Finally, based on the correct repair localization, we evaluate the effectiveness of the repair rules, which represents the correctness of code transformation.

RQ2.1: How accurate is the XGBT model of sGuard+, when compared with state-of-the-art tools?

Results of XGBT. Table 14 shows the comparison of XGBT of sGuard+ and four state-of-the-art nonML-based tools (Slither, Securify, Mythril and sGuard) on PVD. XGBT's F1 scores perform the best among the four tools, except for USI and REN. In particular, for IOU, XGBT has the best F1 score (0.60), and its recall (0.72) is much higher than other tools. Although the precision (0.51) is not as high as other vulnerability types, more FPs do not affect the effectiveness of the

Table 14. The Effect Comparison of XGBT Model and Four nonML-Based Tools (Slither, Securify, Mythril and sGuard) on PVD

	Slither			Securify			Mythril			sGuard			XGBT of sGuard+		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
IOU	-	-	-	-	-	-	0.50	0.36	0.42	0.21	0.42	0.28	**0.51**	**0.72**	**0.60**
UCR	**1.00**	**0.96**	**0.98**	**1.00**	0.95	**0.98**	0.89	0.76	0.82	-	-	-	**1.00**	**0.96**	**0.98**
USI	0.91	**1.00**	**0.95**	-	-	-	**1.00**	0.80	0.89	-	-	-	**1.00**	0.70	0.82
REN	0.81	**0.97**	0.88	**1.00**	0.94	**0.97**	0.45	0.94	0.61	0.45	0.32	0.38	**1.00**	0.90	0.95
TXO	**1.00**	0.72	0.84	-	-	-	**1.00**	**0.94**	**0.97**	**1.00**	**0.94**	**0.97**	**1.00**	**0.94**	**0.97**

[a]- denotes the vulnerability type unsupported by the tool.
[b]The highest numbers are marked in bold.

Table 15. The Effect Comparison of XGBT Model and ML-Based Tools (TMP/DR-GCN and Eth2Vec) on PVD

	TMP/DR-GCN			TMP/DR-GCN + LRD			Eth2Vec			Eth2Vec + LRD			XGBT of sGuard+		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
IOU	-	-	-	-	-	-	0.09	0.30	0.14	#	#	#	**0.51**	**0.72**	**0.60**
UCR	-	-	-	-	-	-	-	-	-	-	-	-	**1.00**	**0.96**	**0.98**
USI	-	-	-	-	-	-	-	-	-	-	-	-	**1.00**	**0.70**	**0.82**
REN	0.91/0.95	0.32/0.67	0.48/0.78	**1.00/1.00**	0.48/0.73	0.65/0.85	0.00	0.00	0.00	#	#	#	**1.00**	**0.90**	**0.95**
TXO	-	-	-	-	-	-	-	-	-	-	-	-	**1.00**	**0.94**	**0.97**

[a]- denotes the vulnerability type unsupported by the tool.
[b]The highest numbers are marked in bold.
[c]# denotes the result that cannot be obtained because the model failed to retrain on the training set LRD.
[d]Note that TMP/DR-GCN [121] proposes two models, namely TMP and DR-GCN, which are evaluated separately.

repair but increase the gas overhead. For UCR and TXO, XGBT has the same best F1 score (0.98, 0.97) as the other tools. For REN, the differences between the F1 score (0.95) of XGBT and the best (0.97) are slight (0.02). For USI, although the F1 score of Slither is the best, its precision (0.91) is the lowest, indicating that it generates many false positives. Mythril has better recall (0.81) than XGBT (0.70), but requires enormous time for detection. XGBT achieves the best precision (1.00), but its recall is lower than the other tools.

Table 15 shows the comparison of the XGBT of sGuard+ and two state-of-the-art ML-based tools (TMP/DR-GCN and Eth2Vec). We separately retrain the models on the dataset from previous work and the training set LRD from this article. Subsequently, we evaluate the effectiveness of the models on our test set PVD. Note that, due to Eth2Vec's inability to handle massive contracts in LRD, we are unable to obtain the retrained model on LRD and the corresponding evaluation results. Additionally, we find that the F1 score of DR-GCN (0.78) is higher than TMP (0.48) on PVD, although TMP outperforms DR-GCN in Reference [121], which indicates that DR-GCN has better generalization. In general, XGBT of sGuard+ outperforms other tools on every type of vulnerability, and our tool supports a wider range of vulnerability types. In particular, the TMP/DR-GCN trained on LRD outperforms the TMP/DR-GCN trained on the original dataset both in precision and recall, indicating that our manually labeled dataset is of higher quality than the dataset of TMP/DR-GCN.

RQ2.2: Do the XGBT model of sGuard+ reduce false positives, and are the Pre-Fs helpful?

Results of FPs. Table 16 shows the evaluation results of FPs reported by four rule-based tools, two ML-based tools and sGuard+ on PVD-R. Each cell represents the number of FPs for functions where the original vulnerability is repaired on PVD-R. Moreover, we find that sGuard has 91 FPs of IOU for the SafeMath functions in the patch code. The reason is that sGuard uses a loose definition

Table 16. The False Positives Due to Ignoring Vulnerability Defense Methods on PVD-R

Vulnerability Types	Slither	Securify	Mythril	sGuard	TMP/DR-GCN	TMP/DR-GCN + LRD	Eth2Vec	Eth2Vec + LRD	XGBT of sGuard+
IOU	-	-	1/121	0/121+91	-	-	19/121	#	0/121
UCR	0/67	0/67	0/67	-	-	-	-	-	0/67
USI	2/10	-	0/10	-	-	-	-	-	0/10
REN	30/31	27/31	29/31	10/31	(14/21)/31	(3/22)/31	0/31	#	0/31
TXO	0/18	-	0/18	0/18	-	-	-	-	0/18
Total	32/126	27/98	30/247	10/170+91	(14/21)/31	(3/22)/31	19/152	#	0/247

[a]- denotes the vulnerability type unsupported by the tool.

[b]# denotes the result cannot be obtained because the tool failed to execute.

[c]Note that TMP/DR-GCN [121] proposes two models, namely TMP and DR-GCN, which are evaluated separately.

of IOU symbolic traces without any checks about preventive conditions. Mythril also reports an FP of IOU since the function has an array out of bounds problem, which is out of the scope of IOU to repair. For REN, the number of FP reported by the four rule-based models on PVD-R is the same as the number of TPs on PVD, which means that all four tools ignore the preventive methods for REN, especially the mutex method. There are no FPs for UCR, USI, and TXO because the repair strategy makes the critical features of vulnerability disappear, such as the repair strategy for TXO replacing `tx.origin` with `msg.sender`.

For ML-based tools, firstly, although DR-GCN has a higher recall rate (0.67–0.32) than TMP on PVD as shown in Table 15, DR-GCN has more FPs (21–14) than TMP on PVD-R as shown in Table 16, which indicates that TMP is sensitive to minor changes in the code and therefore loses generalization, whereas DR-GCN is the opposite of TMP. The `TMP` trained on LRD has fewer FPs (3–14) compared with the `TMP` trained on the original dataset, further emphasizing the importance of reducing false positive labels through manual data confirmation. Secondly, Eth2Vec has the highest number of FPs for IOU, and although it does not have FPs for REN, this could be due to its lack of generalization since it also does not have TPs as shown in Table 15. Note that, as mentioned in RQ2.1, we cannot obtain its evaluation results on PVD-R because Eth2Vec cannot train on LRD. Overall, sGuard+ has no FPs on PVD-R since the Pre-Fs are designed to reduce FPs.

Results of Ablation Experiment. As shown in Table 17, to evaluate the role of Pre-Fs in reducing FPs, we conduct ablation experiments that compared the effectiveness of models with and without Pre-Fs on PVD and PVD-R. As shown in the FP column of Table 17, the effect of Pre-Fs in reducing FPs is more pronounced on PVD-R than on PVD, because the difference between the repaired functions in PVD-R and the corresponding vulnerable functions is subtle, making it easier to mislead models without Pre-Fs.

RQ2.3: Can the localization algorithm find where the true positive vulnerability should be fixed?

Results of Localization Algorithm. We evaluate the effectiveness of the localization algorithm that is the second key step of sGuard+ by comparing it with sGuard on PVD. The vulnerability location error occurs if a vulnerable function is correctly reported by the ML-based detection step, but not repaired correctly. As shown in Table 18, we find that there are 51 TPs of IOU of sGuard, while 23 of them are not properly repaired. Figure 9 shows a typical example that is not repaired thoroughly by sGuard. The arithmetic expression on line 13 is missed by the localization of sGuard. The main reason is that the heuristic applied by sGuard to repair the IOU is insufficient. In contrast, each TP reported by XGBT of sGuard+ is correctly localized by the localization algorithm, because the algorithm is designed to find the location that should be patched according to different types of vulnerability. Furthermore, the main reason for avoiding the errors of sGuard in IOU is that the localization algorithm traverses each statement based on the AST node to repair each vulnerable arithmetic expression.

Table 17. Ablation Experiment of Vulnerability Prevention Features (Pre-Fs)

Dataset	Vulnerability	Features	P	R	F1	TP	FP	TN	FN
PVD	IOU	Without Pre_Fs	0.46	0.74	0.56	89	106	1,339	32
		With Pre_Fs	0.51	0.72	0.60	87	**82**	1,363	34
	UCR	Without Pre_Fs	1	0.96	0.98	64	0	310	3
		With Pre_Fs	1	0.96	0.98	64	0	310	3
	USI	Without Pre_Fs	1	0.9	0.95	9	0	85	1
		With Pre_Fs	1	0.7	0.82	7	0	85	3
	REN	Without Pre_Fs	0.96	0.87	0.92	27	1	138	4
		With Pre_Fs	1	0.9	0.95	28	**0**	139	3
	TXO	Without Pre_Fs	1	0.44	0.62	8	0	25	10
		With Pre_Fs	1	0.94	0.97	17	0	25	1
PVD-R	IOU	Without Pre_Fs	-	-	-	-	**39**	1,481	-
		With Pre_Fs	-	-	-	-	**36**	1,484	-
	UCR	Without Pre_Fs	-	-	-	-	**3**	376	-
		With Pre_Fs	-	-	-	-	**0**	379	-
	USI	Without Pre_Fs	-	-	-	-	**3**	90	-
		With Pre_Fs	-	-	-	-	**0**	93	-
	REN	Without Pre_Fs	-	-	-	-	**24**	138	-
		With Pre_Fs	-	-	-	-	**0**	162	-
	TXO	Without Pre_Fs	-	-	-	-	1	42	-
		With Pre_Fs	-	-	-	-	**0**	43	-

[a]- denotes that the metrics cannot be computed because PVD-R contains only non-vulnerable functions.

Table 18. The Results of TPs of Detection, Correct Localization and Code Transformation by sGuard and sGuard+ on PVD

Vulnerability Types	sGuard			sGuard+		
	TPs	Correctly Localize	Correctly Transform	TPs	Correctly Localize	Correctly Transform
IOU	51	28/51 (55%)	25/51(49%)	85	85/85 (100%)	85/85 (100%)
UCR	-	-	-	64	64/64 (100%)	64/64 (100%)
USI	-	-	-	7	7/7 (100%)	7/7 (100%)
REN	10	10/10 (100%)	9/10 (90%)	28	28/28 (100%)	28/28 (100%)
TXO	17	17/17 (100%)	17/17 (100%)	resp17	17/17 (100%)	17/17 (100%)

[a]- denotes the vulnerability type unsupported by the tool.

RQ2.4: Do the repair rules of sGuard+ ensure that the transformed code is syntactically correct?

Results of Repair rules. As shown in Table 18, the third columns below the second and third columns represent the number of correct code transformations based on correct localization, which evaluate the correctness of the repair rules of sGuard and sGuard+ on PVD, respectively. First, in contracts repaired by sGuard, 6 contracts with IOU fail to compile. Among them, the 3 IOUs in each of the 3 contracts are correctly localized. The main reason for these compilation errors is that sGuard uses string conversion to insert patch code so that the correctness of the syntactic structure of the code cannot be guaranteed. However, the code transformation that implements the repair rules of sGuard+ inserts patch code by modifying the AST nodes and then converting the AST to source code. Therefore, there are no compilation errors in the contracts repaired by sGuard+.

```
contract sGuard{
    function add_uint256(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;}
  ... }
contract token  is sGuard {
    uint256 public totalSupply;
    mapping (address => uint256) public balanceOf;
    ...
    function mintToken(address target, uint256 mintedAmount) onlyOwner {
        balanceOf[target] = add_uint256(balanceOf[target], mintedAmount); //repaired by sGuard
        totalSupply += mintedAmount; //missed by the localization of sGuard
        Transfer(0, this, mintedAmount);
        Transfer(this, target, mintedAmount);}
    ... }
```

Fig. 9. An IOU example from CVE-2018-13087, which is not repaired thoroughly due to the localization problem of sGuard.

Then, as we mentioned in Section 3.1.1, there is a REN repaired by sGuard that changes the original business logic because the repair rule of sGuard for REN adds the same mutex to each vulnerable function even if there are interprocedural calls between functions. It is a significant problem of sGuard, because the high false positive rate of sGuard is likely to cause this problem to occur. Since sGuard+ improves the repair rules for REN based on ICG to address this problem and the vulnerability detection of sGuard+ is more accurate than sGuard, the same problem disappears in the repaired contracts of sGuard. Last, the repair rules extended by sGuard+ for UCR and USI perform well in the vulnerabilities on PVD because their patches for each vulnerable function are independent of each other and the code changes are minor.

Answer to RQ2: First, the XGBT model of sGuard+ achieves the almost best F1 score overall across five vulnerability types on PVD, and significantly eliminates FPs on PVD-R, compared to state-of-the-art tools. Second, the localization of sGuard+ outperforms sGuard especially for IOU on PVD. Third, the Pre-Fs we designed have a significant effect in reducing FPs. Finally, the repair rules of sGuard+ eliminate the limitations of sGuard, and sGuard+ extend simple and robust repair rules for new types of vulnerability. Overall, sGuard+ significantly improves sGuard in terms of vulnerability detection, localization, and repair.

7.4 RQ3: How Efficient is sGuard+ and How Much Gas Overhead of the Patches is Introduced by sGuard+?

To answer the RQ3, we evaluate the performance of sGuard+ on PVD and SRCD compared with sGuard and SCRepair. The timeout of sGuard+ and sGuard is set to be five minutes for each contract the same as [75], and the timeout of SCRepair is one hour in its experiments [115]. Three tools have a memory limit of 4GB, which is the default for Python or Node.js applications on a 64-bit system. Furthermore, since the patch code for repair affects the gas consumption of the transactions of deploying the contract or calling the repaired functions, we compare the difference in gas consumption of these two transaction types before and after the repair on PVD by reproducing transactions.

RQ3.1: What is the time and memory performance of sGuard+?

Time and Memory Consumption. Table 19 shows the evaluation of the time and memory consumption results on PVD with 234 contracts and SCRD with 17 contracts, respectively. In the first

Table 19. Time and Memory Consumption of Three Tools

		PVD				SCRD			Total
	SLoC	0-100	101-200	201-300	>300	0-200	201-400	>400	
SCRepair	OOT	-	-	-	-	0	0/6	4/8 (50%)	4/17 (24%)
	OOM	-	-	-	-	0	0	0	0
	AVG Time	-	-	-	-	660/2=220s	5,520/6=920s	19,380/8=2422.5s	25,560/17=1503.53s
sGuard	OOT	0	1/59 (2%)	3/27 (11%)	3/12 (25%)	0/3	1/6 (17%)	1/8 (13%)	9/188 (4.8%)
	OOM	0	0	0	0	0	1/6 (17%)	3/8 (38%)	4/188 (2.1%)
	AVG Time	54/73=0.74s	1,055/59=17.88s	983/27=36.41s	1,033/12=86.04s	22/3=7.33s	399/6=66.5s	513/8=64.13s	4,059/188=21.59s
sGuard+	OOT	0	0	0	0	0	0	0	0
	OOM	0	0	0	0	0	0	0	0
	AVG Time	259/116=2.23s	295/68=4.34s	175/31=5.65s	239/19=12.58s	11/3=3.67s	41/6=6.83s	134/8=16.75s	1,154/251=4.60s

[a]OOT denotes out of time.
[b]OOM denotes out of memory.
[c]- denotes the vulnerability type not supported by the tool.

Table 20. The Training Time of the XGBT Model of sGuard+

	IOU	UCR	USI	REN	TXO	Average
Training Time (Second)	84s	51s	79s	36s	161s	82.2s
Training Data Size (Thousand)	260k	250k	270k	220k	270k	254k

cell, the physical **source lines of code** (**SLoC**) count the line numbers in the text of the contract source code including the comment and blank lines, as in [84]. To demonstrate the performance of each tool as the number of physical SLoC increases, we divided PVD and SCRD according to the number of SLoC with interval lengths of 100 and 200, respectively, because the average SLoC of SCRD is approximately twice that of PVD.

sGuard+ has the best performance on time consumption with an average time of 4.60 seconds, while sGuard and SCRepair need 21.59 seconds and 25 minutes, respectively. The time performance of sGuard and SCRepair is more significantly affected by the SLoC of the contracts than sGuard+. For example, there are 4 contracts repaired by SCRepair with more than 400 SLoC in SCRD out of time, as well as 7 repaired by sGuard out of time in PVD and 2 contracts in SCRD. According to the range to which the SLoC of the timeout contracts belongs, it is clear that the higher the SLoC of the contracts, the more likely sGuard and SCRepair to have a timeout when repairing the vulnerable contract.

In addition, there are four contracts repaired by sGuard on SCRD out of memory, while none of the contracts repaired by sGuard+ and SCRepair on PVD and SCRD exceed the memory limit. The reason is that the symbolic execution approach performed by sGuard suffers from the path explosion problem that makes the time complexity grow exponentially.

Training Time Consumption. Taking into account the additional overhead of model training, we evaluated the training time of the XGBT model of sGuard+ on each vulnerability type. As shown in Table 20, the training time of the model in the five types of vulnerability is not longer than 3 minutes, among which the shortest training time of the model for IOU is only 24 seconds, and the longest training time of the model for TXO is 161 seconds. In addition, the third row of Table 20 also shows the size of the training data, and we can find that in general, the more data, the longer the training time. On average, about 254,000 data take about 82.2 seconds to train.

RQ3.2: Does sGuard+ reduce the gas overhead compared to sGuard?

Gas Consumption. To evaluate the gas consumption increased by the repair of sGuard+, we compare the difference in gas consumption of the transaction deploying the target contract and calling the target function before and after the repair on PVD. The target function is the public vulnerable function that is repaired correctly by sGuard+ on PVD and the target contract is the

Table 21. The Average Gas Consumption on PVD

Tools	Transaction Types	Contracts	IOU	UCR	USI	REN	TXO	Total
sGuard	Deploy Contracts	Vulnerable	1,494,271	-	-	234,429	244,572	792,411
		Repaired	1,654,655	-	-	300,907	244,572	876,275
		Overhead	160,384 (10.7%)	-	-	66,479 (28.4%)	0 (0.0%)	83,865 (10.6%)
sGuard	Call functions	Vulnerable	35,985	-	-	26,861	27,967	31,062
		Repaired	37,184	-	-	39,615	27,967	34,053
		Overhead	1,200 (3.3%)	-	-	12,755 (47.5%)	0 (0.0%)	2,991 (9.6%)
sGuard+	Deploy Contracts	Vulnerable	1,251,173	469,703	983,450	328,670	244,572	843,723
		Repaired	1,288,398	490,112	1,052,574	353,159	244,572	873,182
		Overhead	37,225 (3.0%)	20,409 (4.3%)	69,124 (7.0%)	24,490 (7.5%)	0 (0.0%)	29,459 (3.5%)
sGuard+	Call functions	Vulnerable	31,365	22,057	43,812	87,876	27,967	40,196
		Repaired	31,604	22,331	45,911	95,528	27,967	41,595
		Overhead	239 (0.8%)	274 (1.2%)	2,098 (4.8%)	7,652 (8.7%)	0 (0.0%)	1,399 (3.5%)

[a]- denotes the vulnerability type unsupported by the tool.

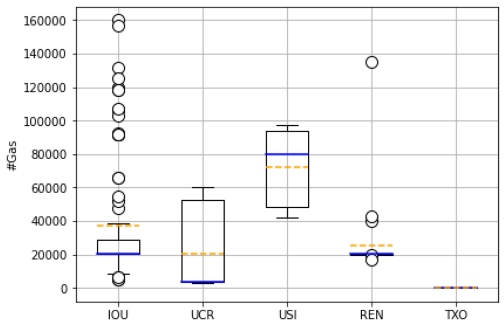

Fig. 10. The extra gas consumption for deploying contracts.

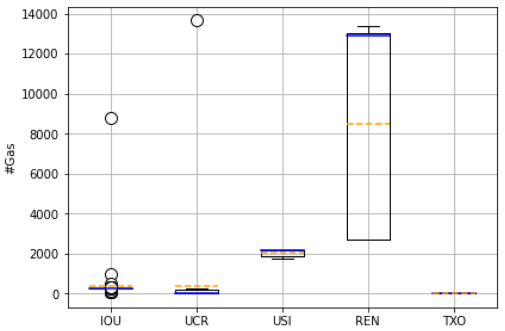

Fig. 11. The extra gas consumption for calling functions.

contract that contains a target function on PVD. In our experiment, the sandbox testnet of Remix, which is the London fork of Ethereum, is used to simulate initiating transactions. The transaction settings (e.g., account, gas limit, value, etc.) and arguments remain the same for the original and repaired contacts or functions. Note that due to the difference between the testnet and the Ethereum environment (e.g., the block number, timestamps and existing data), some transactions to call functions cannot be executed successfully, such as the transaction calling a function that has an external call to a contract with a fixed address on Ethereum but not on the testnet.

Table 21 shows the results of the average gas consumption on PVD. On average, users have to pay an additional 10.6% gas with sGuard and 3.5% gas with sGuard+ to deploy repaired contracts, as well as 9.6% gas with sGuard and 3.5% gas with sGuard+ for calling repaired functions. The gas overhead of sGuard is much higher than that of sGuard+ because the large number of FPs of sGuard (Table 14 shows the low precision of sGuard) introduces many redundant patch code. The extra gas consumption caused by the repair is significantly different for each type of vulnerability because it is directly related to the patch code for each type of vulnerability.

Figures 10 and 11 show the extra gas consumption for deploying contracts and calling functions, respectively. The yellow dotted line represents the average gas consumption. The patch code of USI requires the most additional gas consumption for deploying contracts, and the patch code of REN requires the most additional gas consumption to call functions. In contrast, the patch code of TXO requires the least for both deploying contracts and calling functions. The reason is that the patch code of USI introduces a state variable `__owner` and initializes it as `msg.sender` in the constructor that will be executed when the contract is deployed, and the patch code of REN introduces the

expensive read and write operators in the function to access the mutex state variable, whereas the patch code of TXO merely replaces the variable `tx.origin` to `msg.sender` and accessing these two global variables costs the same amount of gas.

Answer to RQ3: In terms of time performance, sGuard+ (4.60s) performs better than sGuard (21.59s) and SCRepair (1503.53 s), and the training time of the XGBT of sGuard+ takes an average of 82.2 seconds. In terms of memory performance, sGuard+ and SCRepair outperform sGuard that runs out of memory on 24% contracts. The main reason is that ML-based vulnerability detection and rule-based vulnerability repair on sGuard+ have less time consumption than symbolic execution trace analysis on sGuard and search-based repair on SCRepair, respectively. On average, the gas overhead of deploying contracts and calling functions of sGuard+ decreased 7.1 pp (from 3.5% to 10.6%) and 6.1 pp (from 3.5% to 9.6%), respectively, compared to sGuard.

8 DISCUSSION

In this section, we first discuss the threats to validity in Section 8.1. Internal validity refers to trustworthy in terms of the structure and variables of our study, and external validity refers to the universality of the results of our study. Then, we discuss the differences between rule-based and search-based vulnerability repair for smart contracts in Section 8.2. Third, we discuss the necessity for vulnerability detection guided by machine learning rather than voting by off-the-shelf tools in Section 8.3. Finally, we discuss further work in Section 8.5.

8.1 Threats to Validity

Internal Validity. In this work, we adopt three state-of-the-art vulnerability detection tools (Slither, Securify and Mythril) to label the training set. Since existing tools report significantly inconsistent results and a large number of false positives, the results of three tools need to be manually confirmed. However, considering the limited labor costs, we selectively confirm the results of three tools by three authors of this article over a period of two months. The labeling results can be extended in the future with more efficient tools. In addition, we utilize a random search approach for hyperparameter optimization in machine learning models because randomly chosen trials are more efficient than grid search [15]. The number of iterations is a key parameter of random search, which determines the size of the extracted subparameter space. Due to the limitation of computing resources, we set 100 iterations with 10-fold cross-validation to get the optimal parameter combination on the discrete parameter space.

External Validity. The PVD is collected as a ground truth on five vulnerability types to evaluate sGuard+ in our experiments, which has a limited number of contracts and many demonstration contracts without complex contexts compared with real-world contracts. To address the issue of the small size of the evaluation dataset (PVD), we scale up the evaluation on the SCRD of which average SLoC is four times that of PVD since the contracts in it are all from real-world projects. Additionally, for regression testing, the top 350 historical transactions of each contract are collected, which contains all historical transactions of three quarters of the contracts in the dataset (PVD and SCRD). However, due to the difference between the Ethereum network and the local network, historical transactions that rely on environmental information (e.g., block number and timestamp) cannot be reproduced successfully. However, 53% (4,206/7,982) and 54% (9,109/17,235) of transactions are reproduced successfully in our local network for contracts repaired by sGuard and sGuard+, respectively, which can be used as a benchmark to evaluate the impact of repair on the original business logic of contracts.

8.2 Rule-Based Repair or Search-Based Repair?

The rule-based **automated program repair** (**APR**), also known as template-based APR [63], is performed by sGuard and sGuard+, and the search-based APR is performed by SCRepair. Rule-based APR employs predefined program transformation rules as templates to repair for each vulnerability type. Search-based APR employs randomly mutate operators to generate enormous repair candidates, which is usually guided by a heuristic search strategy to find the optimal solution. Rule-based APR is more advantageous than search-based APR to repair vulnerabilities that require complex code transformations with less time consumption, while search-based APR is more general because the patches generated by mutation are flexible for each vulnerability type [42].

For most smart contract vulnerabilities that cause significant asset loss in the real world, the main reason for being exploited is the lack of proper conditional controls [73]. However, the advantage of the search-based approach is to flexibly repair bugs caused by small syntactic errors. Hence, as shown in the experimental results of SCRepair [115], most UCRs are repaired because repairing UCR only requires insertion of a check of the call return value. Whereas none of the IOUs are repaired because the general way to repair IOU is to replace vulnerable arithmetic operators with safe arithmetic functions inherited from the SafeMath contract, which affects many statements outside the vulnerable code block.

Therefore, even though the search-based approach is general, for most explicit vulnerabilities that have fixed repair patterns in smart contracts, the rule-based approach is more effective and efficient than the search-based approach, especially for patterns that require code changes to multiple lines. In addition, the gas overhead introduced by the rule-based approach is more manageable than that introduced by the search-based approach for each vulnerability type.

8.3 Machine Learning Based Guidance or Tools Voting Based?

As we mentioned before, although the existing smart contract vulnerability detection works are full-fledged, there is no reasonable way to combine the existing work to accurately guide vulnerability repair, because the existing tools have significantly inconsistent results for each vulnerability type. As shown in the empirical study [32], there are a considerable number of false positives reported by existing tools and only a small number of vulnerabilities are reported simultaneously by four or more tools. Consequently, we have to face the dilemma of combining tools to guide vulnerability repair.

If we combine tools with high recall rate to cover as many vulnerabilities as possible, there will be a large number of false positives so that the vulnerability repair is prone to affect the non-vulnerable code semantics and generates many unnecessary patches increasing gas overhead. Additionally, if we only trust the vulnerabilities reported by multiple tools simultaneously, there will be a non-negligible number of false negatives because one tool may be complementary to others. For example, Figure 12 shows the REN detection results of Slither, Securify, and Mythril after removing the functions reported repeatedly. The intersection of the three tools is only 0.48% (380/79,397), and not all of them are TPs because the tools have common problems such that the FPs are caused by ignoring the REN defense method of mutexes. Hence, it is infeasible to adopt multiple tools to vote for true vulnerability.

Moreover, considering the execution costs of existing tools, we evaluate whether the performance of sGuard is acceptable. As shown in Figure 13, on URD, most contracts have SLoC between 100 and 400 and there are 23% contracts with more than 400 SLoC, and 514 contracts with more than 2,000 SLoC or even tens of thousands. The average SLoC of contracts is 340. It is known that the ability to analyze these complex contracts is non-trivial for many large smart contract projects in the real world. However, because sGuard may run out of time for contracts with more

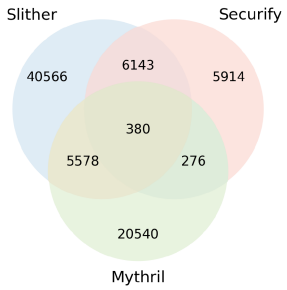

Fig. 12. The REN detection results of Slither, Securify and Mythril on URD.

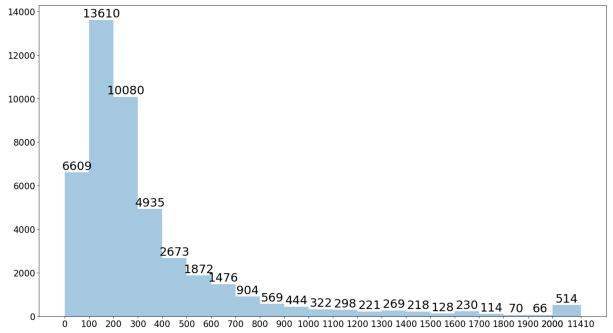

Fig. 13. The SLoC distribution of smart contracts on URD.

than 100 SLoC (see details in Table 19) and the time consumption of the symbolic execution approach (e.g., Mythril and sGuard) grows exponentially due to the path explosion problem, many contracts will not be repaired successfully. On the contrary, the time consumption of the ML-based approach grows linearly with the complexity of the contract. Hence, the ML-based approach has substantial advantages in terms of performance compared with the rule-based approach.

8.4 Limitations of sGuard+

In the evaluation of the XGBT of sGuard+, two main limitations of the machine learning approach are demonstrated: (1) The models do not perform equally well for every type of vulnerability. (2) The models exhibit a tradeoff between recall and precision.

For the first limitation, as shown in Table 14 of RQ2.1, the F1 score (0.60) for IOU is significantly lower than the score (> 0.80) for other vulnerability types. The primary reason for this limitation is that detecting IOU relies on complex arithmetic and logical analysis, which is so challenging that none of the existing tools perform very well. In general, the most suitable approach for detecting IOU is symbolic execution (e.g., Mythril). However, Mythril requires a significant amount of time to detect vulnerabilities, leading to a situation where its practical effectiveness falls short of its theoretical potential. Despite this, the XGBT of sGuard+ outperforms state-of-the-art tools for detecting IOU vulnerability, evening better than Mythril. Additionally, as shown in Table 16 of RQ2.2, FPs for IOU cannot be completely eliminated because the root cause of IOU stems from arithmetic logic, which poses an inherent challenge for ML-based methods relying on static features.

For the second limitation, as shown in Table 14 of RQ2.1, the XGBT of sGuard+ achieves the best precision (1.00) for detecting USI, but its recall is lower than the other tools. The root cause of the low recall of XGBT is mainly attributed to its sensitivity to conditional statements before self-destructing, and BiAn [116] introduces numerous tautologies as branch conditions for obfuscation in PVD, which mislead the model. However, note that conditional statements often serve as effective access control methods for destructing the contract, and tautologies are rarely present in real-word contracts because they not only have no practical effect but also increase gas overhead.

Additionally, as shown in Table 16 of RQ2.2, in the evaluation results on PVD, particularly for the IOU and USI vulnerabilities, we observe an inevitable tradeoff between precision and recall in the models with and without Pre-Fs. For example, compared to the model without Pre-Fs for IOU, the model with Pre-Fs reduces 24 (106-82) FPs, but also incurred a loss of 2 (89-87) TPs. However, the number of TPs sacrificed is negligible compared to the significant reduction in FPs, which is immensely helpful in reducing the gas overhead for the smart contract repair task. Furthermore,

in the evaluation results on PVD-R, the models with Pre-Fs reduce the number of FPs for UCR, USI, REN, and TXO to 0, demonstrating the significant effect of the Pre-Fs.

8.5 Future Work

In recent years, Ethereum is going through a rapid development phase, making the environment of smart contracts dramatically change and the data of smart contracts explodes. For example, Ethereum undergoes at least 2 hard forks per year on average [35], which may change the semantics of the opcode. The `Solidity` language is updated with breaking changes 4 times (from version 0.4.0 to version 0.8.0) [6], which makes significant changes to the syntax and semantics of some of the source code. The number of verified contracts on Etherscan is basically increasing at a rate of more than 500 per day [37], which represents the rapid generation of huge amounts of new smart contract data.

Rapid and complex changes pose great challenges to ML-based vulnerability detection approaches. Since the accuracy and performance of the ML model will decrease as the environment and data change, it is necessary to account for the possible model drift problem [103, 108]. However, even the latest related research works [22, 23] have not yet caught up with the development frontier, and many works stay within a fixed range of applicability as shown in Reference [119]. Therefore, considering the challenge of exploring when models should be re-trained based on the impact of the environment and data changes, we are committed to dedicating efforts to further investigation and addressing this issue in our future research.

9 RELATED WORK

In this section, we first introduce four off-the-shelf APR approaches. Then, the application of these approaches in smart contract vulnerability repair is introduced from the bytecode-level to the source-code-level. Finally, we show existing tools that utilize different program analysis and testing approaches to detect vulnerabilities in smart contracts.

9.1 Automated Program Repair

APR techniques are proposed to alleviate manual effort to identify and remove vulnerabilities in programs [86, 91, 109]. There are two steps of APR: localization and fix generation. Repair can be successful only if the localization is correct and the granularity of the localization is the same as repair [48]. Therefore, it is important to detect vulnerabilities with high accuracy for a repair tool, and the appropriate granularity should be localized according to the repair rules.

Generate-and-validate and semantics-driven repairs are two main approaches on APR [42]. *Generate-and-validate* approach repair programs by generating potential solutions until a solution is successfully validated. Heuristics, constraint solving, and machine learning are three main techniques used to drive patch generation [64]. In addition, patches can be generated based on fix patterns or templates [53].

- **Heuristic search** technique is used to guide patch mutations. GenProg [57] uses **genetic programming** (**GP**) to search a solution passing all test cases from randomly mutated variants. RSRepair [87] uses random search rather than GP to guide the patch generation process, which is more effective and efficient than GenProg.
- **Constraint solving** technique utilizes an SMT solver to address the problem of program synthesis. SemFix [74] formulates the requirement passing tests as a constraint by symbolic execution and repairs expressions by program synthesis. Nopol [112] fixes buggy conditional statements by translating the synthesis problem into the SMT problem based on collected runtime traces.

— **Machine learning** technique learns bug-fixing patterns by seq2seq models to transform the bug code into the fixed code. DLFix [59] uses a two-tier DL model in which the first layer learns the context of bug fixes and the second layer learns the bug-fixing code transformations. CURE [52] uses pre-trained model, code-aware search strategy and subword tokenization technique to improve the effectiveness of repair.
— **Template-based** APR approach uses fix patterns collected by a human study. PAR [53] is proposed to reduce non-sensical patches generated by random mutations, using fix patterns learned from existing human-writing patches. TBar [63] extends fix patterns collected from the data in the literature.

sGuard+ repairs specific vulnerability types based on the corresponding existing public repair strategies. Repair rules are designed to patch the vulnerable source code on typical patterns with different contexts. The main idea of rule design is to prevent the vulnerability from being exploited without breaking the original function with the minimum gas overhead.

9.2 Vulnerability Repair for Smart Contracts

Bytecode-level Repair Tools. SMARTSHIELD [117], EVMPATCH [90], and ELYSIUM [101] are three ARP tools of the bytecode-level for smart contracts. SMARTSHIELD extracts the bytecode-level semantic information by analyzing the AST and EVM byte code of contracts and then repairs the byte code by transforming the control flow and inserting a sequence of data guard instructions. The bytecode-level semantic information of the repaired contract is compared with the original contract to validate that the irrelevant functions are not affected.

EVMPATCH performs a template-based repair approach. EVMPATCH proposes a bytecode rewriting engine to adapt the patch templates to the vulnerable contracts and the transactions of the original contract are automatically replayed to validate the correctness of the repair. The approach of ELYSIUM is the same as SMARTSHIELD and EVMPATCH, but ELYSIUM supports more vulnerabilities and minimizes the runtime gas overhead of transactions than the other tools.

Source-code-level Repair Tools. SCRepair [115] is the first search-based source-code-level APR tool for contracts. Localization of SCRepair is based on two vulnerability detection tools (Slither and Oyente). Move, inset, and replace operators are used to mutate statement-level or expression-level buggy code on the AST of the original vulnerable contract. SCRepair introduces the notion of gas dominance to rank the repaired candidate contracts, and the repair with the lowest gas cost will be selected first. The validity criteria of the generated patches contain three factors: test cases, outsource vulnerability detection tools, and gas consumption bounds.

In contrast to SCRepair whose repair is plausible, sGuard performs a semantic-driven approach on the source code of contracts so that the vulnerability of contracts is guaranteed to be solved correctly. In addition, sGuard is the only tool that localizes vulnerabilities through self-contained detection modules. However, we find that the detection approach of sGuard based on symbolic execution is inaccurate and time-consuming, so the correctness of repair is severely affected, and the patches inserted by sGuard may introduce new vulnerabilities and affect the original business logic.

sGuard+ works on improving sGuard to expand the supported vulnerability types and achieve better effectiveness and efficiency. sGuard+ corrects the weak patch that cannot completely defend against vulnerability and improves the correctness of repair based on the accurate machine learning based detection approach and the corresponding localization algorithm.

9.3 Vulnerability Detection for Smart Contracts

Program analysis and testing contain multiple substantial approaches applied to automatic program vulnerability detection such as static analysis, symbolic execution, and fuzzing. In recent years, researchers have proposed many tools for detecting vulnerabilities in smart contracts.

— **Static analysis** detects vulnerabilities by examining program properties from code abstract representations without executing code. Since all possible conditional branch paths are considered, the static analysis approach has a high recall rate on vulnerability detection, but there may be many false positives caused by infeasible paths [14]. Static analysis approaches are employed by [10, 38, 39, 41, 61, 100, 113] to detect vulnerabilities in smart contracts. Among them, Slither [38] utilizes data flow analysis and taint tracking based on an intermediate representation to detect approximately 20 types of vulnerabilities. Both SmartCheck [100] and SESCon [10] are based on XML parse tree to detect vulnerabilities. SmartEmbed [41] utilizes the code clone approach, which identifies vulnerabilities based on the similarity with existing vulnerable code. Clairvoyance [113] proposes an accurate cross-contract call chain analysis approach at the source-code-level, and summaries five major **path protective techniques** (**PPTs**) to specifically reduce false positives for cross-contract reentrancy vulnerabilities. Compared with PPTs, we propose Pre-Fs to reduce FPs of five general non-cross-contract vulnerability types. Although cross-contract reentrancy vulnerabilities are more difficult to detect than REN, some PPTs are applicable to both cross-contract and general reentrancy vulnerabilities, which inspire the design of the Pre-Fs. For example, we design the `unprotected_ren` feature to recognize PPT4 and PPT5 of Clairvoyance. However, in practice, the design of PPTs is static and limited in number, while the machine learning method is more general and flexible, and the accuracy of the model can be improved by designing richer Pre-Fs. In contrast, SmartDagger [61] focuses on cross-contract vulnerability detection at the bytecode-level. SmartBugs [39] is a static analysis framework that integrates ten tools and two datasets that are used for comparison between tools.

— **Symbolic execution** is a way that uses symbolic variables rather than concrete values as input to simulate program execution so that the feasible path can be determined by solving the constraints. If a feasible path violates pre-defined safety properties, the corresponding vulnerability will be reported. Compared with static analysis, the symbolic execution approach can avoid false positives, but a tradeoff must be made between loop iteration bounds and time consumption due to the path explosion problem [13]. Symbolic execution approaches are used by [17, 21, 26, 67, 70, 83, 102] to detect vulnerabilities in smart contracts. Among them, Securify [102] utilizes symbolic execution on the contract dependency graph to extract semantic data that can be used to check compliance and violation patterns. Mythril [26] and Manticore [70] use concolic execution and dynamic symbolic execution approaches, respectively. SAILFISH [17] combines value-summary analysis with symbolic execution to effectively reduce false positives for state-inconsistency bugs (i.e., reentrancy and transaction order dependence).

— **Fuzzing** is a dynamic testing approach that generates numerous unexpected inputs to repeatedly execute programs and monitors exceptions when the program is executed. In general, fuzzing reveals vulnerabilities without false positives, while the execution cost is expensive and the low code coverage leads to a high false negative rate [120]. Existing work [40, 51, 62, 76, 114] use fuzzing approaches to detect vulnerabilities in smart contracts. Among them, sFuzz [76] utilizes a feedback-guided fuzzing algorithm that transforms the test generation problem into an optimization problem. xFuzz [114] improves sFuzz using a machine learning-guided approach to reduce the search space.

— **Machine learning** has received increasing attention in the last ten years to address the problem of program vulnerability detection [44, 118]. Machine learning can learn vulnerable code patterns from vulnerability samples to recognize similar vulnerabilities. Compared to the conventional approaches mentioned above, the machine learning approach is more general than fixed detection rules because various vulnerability patterns can be learned based on flexible features. Feature engineering is the key factor to achieve high precision and recall for vulnerability detection, and designing features is difficult because they are domain specific [31]. Specifically, rich features that are positively correlated with vulnerabilities can reduce false negatives, and those that are negatively correlated with vulnerabilities can reduce false positives. [12, 107] use machine learning to detect vulnerabilities of smart contracts. Eth2Vec [12] utilizes a neural network of natural language processing to learn features of vulnerable EVM bytecodes and detects vulnerabilities based on code similarity. ContractWard [107] transforms the source code into the opcode for simplification and then extracts 1619 bigram features from the simplified opcodes. ContractWard uses multi-label classification and resampling algorithm to build the model.

sGuard+ utilizes a vulnerability detection approach based on machine learning. Compared to these tools that employ program analysis and testing approaches, sGuard+ is less time-consuming than symbolic execution and fuzzing tools, and sGuard+ reduces false positives reported by static analysis because the features negatively correlated with vulnerabilities are designed to identify various methods of vulnerability prevention. In addition, sGuard+ designs rich features that contain both source code features by static analysis and opcode features by a pre-trained Word2Vec model, while the source code features are ignored by ContractWard and Eth2Vec.

10 CONCLUSION

In this work, we propose a machine learning guided rule-based automated vulnerability repair approach to accurately detect and correctly repair five types of vulnerability in smart contracts. We implement a tool called sGuard+, which extends and enhances sGuard. Our approach addresses three key challenges: (1) Accurately guide vulnerability repair using machine learning models and reduce false positives resulting from ignoring vulnerability prevention strategies. (2) Refine the repair rules of sGuard to preserve the original business logic of the contracts. And (3) For the whole workflow, sGuard+ repairs vulnerabilities with lower time consumption and gas overhead. Experiment results show that sGuard+ is more effective and efficient than existing source-code-level repair tools.

REFERENCES

[1] 2016. DAO at v1.0. Retrieved from https://github.com/blockchainsllc/DAO/tree/v1.0. Online; accessed 17 June 2016.
[2] 2017. The Parity Wallet Hack Explained. Retrieved from https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. Online; accessed 19 July 2017.
[3] 2022. CVE-2020-19765. Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-19765. Online; accessed 1 January 2022.
[4] 2022. The CVE Records Related to Smart Contracts Without Explict Keyworks. Retrieved from https://github.com/ToolmanInside/CVEs. Online; accessed 1 January 2022.
[5] 2022. Etherscan. Retrieved from https://etherscan.io/. Online; accessed 25 April 2022.
[6] 2022. Solidity Documentation. Retrieved from https://docs.soliditylang.org/en/v0.4.26/. Online; accessed 25 April 2022.
[7] 2022. Solidity v0.8.0 Breaking Changes. Retrieved from https://docs.soliditylang.org/en/breaking/080-breaking-changes.html. Online; accessed 25 April 2022.
[8] 2023. The Correctness Evaluation Results of Elysium. Retrieved from https://github.com/gcf3711/truffle_example/tree/main/elysium. Online; accessed 10 Jun 2023.

[9] Rachit Agarwal, Tanmay Thapliyal, and Sandeep K. Shukla. 2021. Vulnerability and transaction behavior based detection of malicious smart contracts. In *International Conference on Cryptography and Security Systems*.

[10] Amir Ali, Zain Ul Abideen, and Kalim Ullah. 2021. SESCon: Secure ethereum smart contracts by vulnerable patterns' detection. *Secur. Commun. Networks* 2021 (2021), 2897565:1–2897565:14. https://www.hindawi.com/journals/scn/2021/2897565/

[11] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2021. Eth2Vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure* 3, 4 (2021), 10010.

[12] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2021. Eth2Vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *BSCI'21: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, Virtual Event, Hong Kong, June 7, 2021*, Keke Gai and Kim-Kwang Raymond Choo (Eds.). ACM, 47–59. DOI: https://doi.org/10.1145/3457337.3457841

[13] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. DOI: https://doi.org/10.1145/3182657

[14] Thomas Ball. 1999. The concept of dynamic analysis. In *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1687)*, Oscar Nierstrasz and Michel Lemoine (Eds.). Springer, 216–234. DOI: https://doi.org/10.1007/3-540-48166-4_14

[15] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.* 13, 10 (2012), 281–305. DOI: https://doi.org/10.5555/2503308.2188395

[16] Gérard Biau and Erwan Scornet. 2016. A random forest guided tour. *Test* 25, 2 (2016), 197–227.

[17] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *43RD IEEE Symposium On Security and Privacy (SP'22)* (IEEE Symposium on Security and Privacy), IEEE COMPUTER SOC, 10662 LOS VAQUEROS CIRCLE, PO BOX 3014, LOS ALAMITOS, CA 90720-1264 USA, 161–178. DOI: https://doi.org/10.1109/SP46214.2022.00072

[18] Chainalysis. 2022. The Chainalysis 2022 Crypto Crime Report. Retrieved from https://go.chainalysis.com/2022-Crypto-Crime-Report.html. Online; accessed 9 Sep 2022.

[19] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. 2022. Deep learning based vulnerability detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (September 2022), 3280–3296. DOI: https://doi.org/10.1109/TSE.2021.3087402

[20] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.* 53, 3 (2020), 67:1–67:43. DOI: https://doi.org/10.1145/3391195

[21] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode. *IEEE Trans. Software Eng.* 48, 7 (2022), 2189–2207. DOI: https://doi.org/10.1109/TSE.2021.3054928

[22] Jiachi Chen, Xin Xia, David Lo, and John C. Grundy. 2020. Why do smart contracts self-destruct? Investigating the selfdestruct function on ethereum. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2020), 1–37.

[23] Jiachi Chen, Xin Xia, David Lo, John C. Grundy, Xiapu Luo, and Ting Chen. 2019. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering* 48, 1 (2019), 327–345.

[24] Tianqi Chen, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, Rory Mitchell, Ignacio Cano, Tianyi Zhou, et al. 2015. Xgboost: Extreme gradient boosting. *R Package Version 0.4-2* 1, 4 (2015), 1–4. https://scholar.google.com/scholar?hl=zh-CN&as_sdt=0%2C5&as_vis=1&q=Xgboost%3A+Extreme+gradient+boosting&btnG=

[25] ConsenSys. 2019. Truffle Framework Documentation. Retrieved from https://trufflesuite.com/docs/truffle/. Online; accessed 29 January 2022.

[26] ConsenSys. 2021. Mythril. Retrieved from https://github.com/ConsenSys/mythril-classic. Online; accessed 12 October 2021.

[27] DHS and CISA. 2022. CVE Website. Retrieved from https://cve.mitre.org/. Online; accessed 1 January 2022.

[28] Monika di Angelo and Gernot Salzer. 2019. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON'19)*. 69–78. DOI: https://doi.org/10.1109/DAPPCON.2019.00018

[29] Bruno Dia, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro. 2021. An empirical evaluation of the effectiveness of smart contract verification tools. In *26th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2021, Perth, Australia, December 1–4, 2021*. IEEE, 17–26.https://doi.org/10.1109/PRDC53464.2021.00013

[30] ConsenSys Diligence. 2022. Ethereum Smart Contract Security Best Practices. https://consensys.github.io/smart-contract-best-practices/. Online; accessed 25 April 2022.
[31] Pedro Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM* 55, 10 (2012), 78–87.
[32] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 530–541. DOI: https://doi.org/10.1145/3377811.3380364
[33] Mojtaba Eshghie, Cyrille Artho, and Dilian Gurov. 2021. Dynamic vulnerability detection on smart contracts using machine learning. In *EASE'21*, Association for Computing Machinery, Trondheim, Norway, 305–312. DOI: https://doi.org/10.1145/3463274.3463348
[34] Ethereum. 2022. Decentralized Applications. Retrieved from https://ethereum.org/en/dapps/. Online; accessed 9 Aug 2022.
[35] Ethereum. 2022. History. Retrieved from https://ethereum.org/en/history/. Online; accessed 29 January 2022.
[36] Ethereum. 2022. Yellow Paper. Retrieved from https://ethereum.github.io/yellowpaper/paper.pdf. Online; accessed 25 April 2022.
[37] Etherscan. 2022. Verified Contracts. Retrieved from https://etherscan.io/chart/verified-contracts. Online; accessed 29 January 2022.
[38] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 8–15. DOI: https://doi.org/10.1109/WETSEB.2019.00008
[39] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A framework to analyze solidity smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21–25, 2020*. IEEE, 1349–1352. DOI: https://doi.org/10.1145/3324884.3415298
[40] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. EVMFuzzer: Detect EVM vulnerabilities via fuzz testing. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 1110–1114. DOI: https://doi.org/10.1145/3338906.3341175
[41] Zhipeng Gao, Vinoj Jayasundara, Lingxiao Jiang, Xin Xia, David Lo, and John C. Grundy. 2019. SmartEmbed: A tool for clone and bug detection in smart contracts through structural code embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29–October 4, 2019*. IEEE, 394–397. DOI: https://doi.org/10.1109/ICSME.2019.00067
[42] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic software repair: A survey. *IEEE Trans. Softw. Eng.* 45, 1 (2019), 34–67. DOI: https://doi.org/10.1109/TSE.2017.2755013
[43] GeeksforGeeks. 2022. What was the DAO Hack? Retrieved from https://www.geeksforgeeks.org/what-was-the-dao-hack/. Online; accessed 29 January 2022.
[44] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.* 50, 4 (2017), 56:1–56:36. DOI: https://doi.org/10.1145/3092566
[45] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In *ISSTA'20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18–22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 415–427. DOI: https://doi.org/10.1145/3395363.3397385
[46] Ajay K. Gogineni, Soumya Swayamjyoti, Devadatta Sahoo, Kisor Kumar Sahu, and Raj Kishore. 2020. Multi-class classification of vulnerabilities in smart contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing. *IOP SciNotes* 1, 3 (2020), 035002.
[47] Google. 2022. Bigquery. Retrieved from https://console.cloud.google.com/bigquery?project=ethereal-shape-303507. Online; accessed 25 April 2022.
[48] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 3–13. DOI: https://doi.org/10.1109/ICSE.2012.6227211
[49] H-X. 2022. Top 3 Smart Contract Audit Tools. Retrieved from https://www.h-x.technology/blog/top-3-smart-contract-audit-tools. Online; accessed 9 Aug 2022.
[50] Hui Han, Wenyuan Wang, and Binghuan Mao. 2005. Borderline-SMOTE: A new over-sampling method in imbalanced data sets learning. In *Advances in Intelligent Computing, International Conference on Intelligent Computing, ICIC 2005, Hefei, China, August 23–26, 2005, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 3644)*, De-Shuang Huang, Xiao-Ping (Steven) Zhang, and Guang-Bin Huang (Eds.). Springer, 878–887. DOI: https://doi.org/10.1007/11538059_91

[51] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 259–269. DOI: https://doi.org/10.1145/3238147.3238177

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

[53] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE'13)*. 802–811. DOI: https://doi.org/10.1109/ICSE.2013.6606626

[54] Masanari Kondo, Gustavo Ansaldi Oliva, Zhen Ming Jack Jiang, Ahmed E. Hassan, and Osamu Mizuno. 2020. Code cloning in smart contracts: A case study on verified contracts from the Ethereum blockchain platform. *Empirical Software Engineering* 25, 6 (2020), 4617–4675.

[55] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise static modeling of Ethereum "memory". *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 190:1–190:26. DOI: https://doi.org/10.1145/3428258

[56] George Lawton. 2022. Top 9 Blockchain Platforms to Consider in 2022. https://www.techtarget.com/searchcio/feature/Top-9-blockchain-platforms-to-consider. Online; accessed 9 Aug 2022.

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

[58] Nicolas Lesimple and Martin Jaggi. 2020. *Exploring Deep Learning Models for Vulnerabilities Detection in Smart Contracts*. Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland.

[59] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based code transformation learning for automated program repair. In *ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 602–614. DOI: https://doi.org/10.1145/3377811.3380345

[60] Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. 2019. SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (2019), 458–465.

[61] Zeqin Liao, Zibin Zheng, Xiao Cui Chen, and Yuhong Nan. 2022. SmartDagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022). Retrieved from https://api.semanticscholar.org/CorpusID:250562430

[62] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: Finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27–June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 65–68. DOI: https://doi.org/10.1145/3183440.3183495

[63] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15–19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. DOI: https://doi.org/10.1145/3293882.3330577

[64] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, Yves Le Traon, Tegawendé Bissyandé, and Dongsun Kim. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. DOI: https://doi.org/10.1145/3377811.3380338

[65] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In *International Joint Conference on Artificial Intelligence*. Retrieved from https://api.semanticscholar.org/CorpusID:235458204

[66] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT: Ethereum smart COntRacTs vulnerability detection using deep neural network and transfer learning. arXiv:2103.12607. Retrieved from https://arxiv.org/abs/2103.12607

[67] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 254–269. DOI: https://doi.org/10.1145/2976749.2978309

[68] Na Meng, Stefan Nagy, Danfeng Daphne Yao, Wenjie Zhuang, and Gustavo A. Arango-Argoty. 2017. Secure coding practices in Java: challenges and vulnerabilities. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2017), 372–383. Retrieved from https://api.semanticscholar.org/CorpusID:3480894

[69] Pouyan Momeni, Yu Wang, and Reza Samavi. 2019. Machine learning model for smart contracts security analysis. In *2019 17th International Conference on Privacy, Security and Trust (PST)* (2019), 1–6.

[70] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11–15, 2019*. IEEE, 1186–1189. DOI: https://doi.org/10.1109/ASE.2019.00133

[71] Anthony J. Myles, Robert N. Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. 2004. An introduction to decision tree modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society* 18, 6 (2004), 275–285.

[72] MythX. 2021. SWC Registry. Retrieved from https://swcregistry.io/. Online; accessed 12 October 2021.

[73] NCC Group. 2019. Decentralized Application Security Project (or DASP) Top 10 of 2018. Retrieved from https://dasp.co/. Online; accessed 29 January 2019.

[74] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE'13, San Francisco, CA, USA, May 18–26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 772–781. DOI: https://doi.org/10.1109/ICSE.2013.6606623

[75] Tai D. Nguyen, Long H. Pham, and Jun Sun. 2021. SGUARD: Towards fixing vulnerable smart contracts automatically. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021*. IEEE, 1215–1229. DOI: https://doi.org/10.1109/SP40001.2021.00057

[76] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An efficient adaptive fuzzer for solidity smart contracts. In *ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 778–788. DOI: https://doi.org/10.1145/3377811.3380334

[77] NickLennonLiu. 2023. How to Produce the Graph Feature from Onehot Vectors? Retrieved from https://github.com/Messi-Q/AMEVulDetector/issues/4. Online; accessed 25 October 2023.

[78] Openzeppelin. 2022. Access Control. Retrieved from https://docs.openzeppelin.com/contracts/4.x/access-control. Online; accessed 29 January 2022.

[79] OpenZeppelin. 2022. A Library for Secure Smart Contract Development. Retrieved from https://github.com/OpenZeppelin/openzeppelin-contracts/. Online; accessed 25 April 2022.

[80] Openzeppelin. 2022. SafeMath. Retrieved from https://github.com/binodnp/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol. Online; accessed 29 January 2022.

[81] Openzeppelin. 2022. Security. Retrieved from https://docs.openzeppelin.com/contracts/4.x/api/security#ReentrancyGuard. Online; accessed 29 January 2022.

[82] PeckShield. 2022. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. Retrieved from https://peckshield.medium.com/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09. Online; accessed 9 Sep 2022.

[83] Anton Permenev, Dimitar K. Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. 2020. VerX: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18–21, 2020*. IEEE, 1661–1677. DOI: https://doi.org/10.1109/SP40000.2020.00024

[84] Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Tonelli, and Michele Marchesi. 2019. A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access* 7 (2019), 78194–78213. https://webofscience.clarivate.cn/wos/alldb/full-record/WOS:000473774600001

[85] Kamil Polak. 2022. Hack Solidity: Reentrancy Attack. Retrieved from https://hackernoon.com/hack-solidity-reentrancy-attack. Online; accessed 9 Sep 2022.

[86] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE'14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 254–265. DOI: https://doi.org/10.1145/2568225.2568254

[87] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE'14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 254–265. DOI: https://doi.org/10.1145/2568225.2568254

[88] J. Ross Quinlan et al. 1996. Bagging, boosting, and C4. 5. In *Aaai/Iaai, vol. 1*. 725–730.

[89] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. Empirical evaluation of smart contract testing: What is the best choice?. In *ISSTA'21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11–17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 566–579. DOI: https://doi.org/10.1145/3460319.3464837

[90] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2021. EVMPatch: Timely and automated patching of ethereum smart contracts. In *30th USENIX Security Symposium, USENIX Security 2021, August 11–13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1289–1306. Retrieved from https://www.usenix.org/conference/usenixsecurity21/presentation/rodler

[91] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin C. Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 43–54. DOI: https://doi.org/10.1145/2737924.2737988

[92] Slither. 2022. Control Flow Node. Retrieved from https://github.com/crytic/slither/blob/master/slither/core/cfg/node.py. Online; accessed 29 January 2022.

[93] SmartBugs. 2021. Dataset. Retrieved from https://github.com/smartbugs/smartbugs/tree/master/dataset. Online; accessed 12 October 2021.

[94] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VERISMART: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18–21, 2020*. IEEE, 1678–1694. DOI: https://doi.org/10.1109/SP40000.2020.00032

[95] Solidity Documentation. 2022. Security Considerations. Retrieved from https://docs.soliditylang.org/en/v0.4.26/security-considerations.html#use-the-checks-effects-interactionspattern. Online; accessed 29 January 2022.

[96] Yuhang Sun and Lize Gu. 2021. Attention-based machine learning model for smart contract vulnerability detection. *Journal of Physics: Conference Series* 1820, 1 (2021), 012004.

[97] Onur Sürücü, Uygar Yeprem, Connor Wilkinson, Waleed Hilal, Stephen Andrew Gadsden, John Yawney, Naseem Alsadi, and Alessandro Giuliano. 2022. A survey on ethereum smart contract vulnerability detection using machine learning. In *Defense + Commercial Sensing*.

[98] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997). https://webofscience.clarivate.cn/wos/alldb/full-record/INSPEC:5726368

[99] Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sengupta, and Y. Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. arXiv:1811.06632. Retrieved from https://arxiv.org/abs/1811.06632

[100] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static analysis of ethereum smart contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27–June 3, 2018*, Roberto Tonelli, Giuseppe Destefanis, Steve Counsell, and Michele Marchesi (Eds.). ACM, 9–16. DOI: https://doi.org/10.1145/3194113.3194115

[101] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2022. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In *RAID'22*, Association for Computing Machinery, Limassol, Cyprus, 115–128. DOI: https://doi.org/10.1145/3545948.3545975

[102] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 67–82. DOI: https://doi.org/10.1145/3243734.3243780

[103] Alexey Tsymbal. 2004. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin* 106, 2 (2004), 58.

[104] Gerhard Wagner. 2021. Authorization through tx.origin Vulnerability. Retrieved from https://swcregistry.io/docs/SWC-115. Online; accessed 12 October 2021.

[105] Gerhard Wagner. 2021. EIP-1470. Retrieved from https://eips.ethereum.org/EIPS/eip-1470. Online; accessed 12 October 2021.

[106] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2020. ContractWard: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2020), 1133–1144.

[107] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2021. ContractWard: Automated vulnerability detection models for ethereum smart contracts. *IEEE Trans. Netw. Sci. Eng.* 8, 2 (2021), 1133–1144. DOI: https://doi.org/10.1109/TNSE.2020.2968505

[108] Geoffrey I. Webb, Roy Hyde, Hong Cao, Hai-Long Nguyen, and François Petitjean. 2015. Characterizing concept drift. *Data Mining and Knowledge Discovery* 30, 4 (2015), 964–994.

[109] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16–24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. DOI: https://doi.org/10.1109/ICSE.2009.5070536

[110] Cipai Xing, Zhuo Chen, Lexin Chen, Xiaojie Guo, Zibin Zheng, and Jin Li. 2020. A new scheme of vulnerability analysis in smart contract with machine learning. *Wireless Networks* (2020), 1–10. https://webofscience.clarivate.cn/wos/alldb/full-record/WOS:000546538400002

[111] Yingjie Xu, Gengran Hu, Lin You, and Chengtang Cao. 2021. A novel machine learning-based analysis model for smart contract vulnerability. *Secur. Commun. Networks* 2021 (2021), 5798033:1–5798033:12. https://www.hindawi.com/journals/scn/2021/5798033/

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

[113] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21–25, 2020*. IEEE, 1029–1040. DOI: https://doi.org/10.1145/3324884.3416553

[114] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xFuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022), 1–14. https://ieeexplore.ieee.org/document/9795233?denied=

[115] Xiao Liang Yu, Omar I. Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart contract repair. *ACM Trans. Softw. Eng. Methodol.* 29, 4 (2020), 27:1–27:32. DOI: https://doi.org/10.1145/3402450

[116] Meng Zhang, Pengcheng Zhang, Xiapu Luo, and Feng Xiao. 2020. Source code obfuscation for smart contracts. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC'20)*. 513–514. DOI: https://doi.org/10.1109/APSEC51365.2020.00069

[117] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic smart contract protection made easy. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18–21, 2020*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, 23–34. DOI: https://doi.org/10.1109/SANER48275.2020.9054825

[118] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F. Bissyandé, Jacques Klein, and John C. Grundy. 2021. On the impact of sample duplication in machine-learning-based Android malware detection. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 40:1–40:38. DOI: https://doi.org/10.1145/3446905

[119] Zibin Zheng, Neng Zhang, Jianzhong Su, Zhijie Zhong, Mingxi Ye, and Jiachi Chen. 2023. Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, IEEE Press, Melbourne, Victoria, Australia, 295–306. DOI: https://doi.org/10.1109/ICSE48619.2023.00036

[120] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A survey for roadmap. *ACM Comput. Surv.* 54, 11s (2022), 1–36. DOI: https://doi.org/10.1145/3512345 Just Accepted.

[121] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart contract vulnerability detection using graph neural network. In *International Joint Conference on Artificial Intelligence*.

Received 24 November 2022; revised 15 November 2023; accepted 9 January 2024