VulHunter: Hunting Vulnerable Smart Contracts at EVM bytecode-level via Multiple Instance Learning

Zhaoxuan Li, *Student Member, IEEE,* Siqi Lu, Rui Zhang, Ziming Zhao, Rujin Liang, Rui Xue, *Member, IEEE,* Wenhao Li, *Student Member, IEEE,* Fan Zhang, *Member, IEEE,* and Sheng Gao, *Member, IEEE*

Abstract—With the economic development of Ethereum, the frequent security incidents involving smart contracts running on this platform have caused billions of dollars in losses. Consequently, there is a pressing need to identify the vulnerabilities in contracts, while the state-of-the-art (SOTA) detection methods have been limited in this regard as they cannot overcome three challenges at the same time. (i) Meet the requirements of detecting the source code, bytecode, and opcode of contracts simultaneously; (ii) reduce the reliance on manual pre-defined rules/patterns and expert involvement; (iii) assist contract developers in completing the contract lifecycle more safely, *e.g.,* vulnerability repair and abnormal monitoring. With the development of machine learning (ML), using it to detect the contract runtime execution sequences (called instances) has made it possible to address these challenges. However, the lack of datasets with fine-grained sequence labels poses a significant obstacle, given the unreadability of bytecode/opcode. To this end, we propose a method named VulHunter that extracts the instances by traversing the Control Flow Graph built from contract opcodes. Based on the hybrid attention and multi-instance learning mechanisms, VulHunter reasons the instance labels and designs an optional classifier to automatically capture the subtle features of both normal and defective contracts, thereby identifying the vulnerable instances. Then, it combines the symbolic execution to construct and solve symbolic constraints to validate their feasibility. Finally, we implement a prototype of VulHunter with 15K lines of code and compare it with 9 SOTA methods on five open source datasets including 52,042 source codes and 184,289 bytecodes. The results indicate that VulHunter can detect contract vulnerabilities more accurately (90.04% accurate rate and 85.60% F1 score), efficiently (only took 4.4 seconds per contract), and robustly (0% analysis failed rate) than the SOTA methods. Also, it can focus on specific metrics such as precision and recall by employing different baseline models and hyperparameters to meet the various user requirements, *e.g.,* vulnerability discovery and misreport mitigation. More importantly, compared with the previous ML-based arts, it can not only provide classification results, defective contract source code statements, key opcode fragments, and vulnerable execution paths, but also eliminate misreports and facilitate more operations such as vulnerability repair and attack simulation during the contract lifecycle.

Index Terms—Blockchain, smart contract, security analysis, multiple instance learning, and symbolic execution

1 Introduction

Blockchain and its killer applications, *e.g.,* Bitcoin, Metaverse, and Non-Fungible Token (NFT), are taking the world by storm [1], [2], [3]. **Smart contracts** are programs running on top of the blockchain [3], [4]. Millions of smart contracts have been deployed on the Ethereum blockchain platform, the most popular blockchain network for Web3 and decentralized apps (dApps), enabling a wide range of applications including wallets, crowdfunding, decentralized gambling, online gaming, and cross-industry finance [5].

(Corresponding author: Rui Zhang.)

- *Zhaoxuan Li, Rui Zhang, Rui Xue, and Wenhao Li are with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, 100093, China, and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, 100049, China. E-mail: {lizhaoxuan, zhangrui, xuerui, liwenhao}@iie.ac.cn.*
- *Siqi Lu and Rujin Liang are with the Information Engineering University, Zhengzhou, 450001, China, and Henan Key Laboratory of Network Cryptography Technology, Zhengzhou, 450001, China. E-mail: 080lusiqi@sina.com and coderlrj@163.com.*
- *Ziming Zhao and Fan Zhang are with the Zhejiang University, Hangzhou, 310027, China. Fan Zhang is also with ZJU-Hangzhou Global Scientific and Technological Innovation Center, 311200, with the Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, 310027 and with Jiaxing Research Institute, Zhejiang University, 314000, China. E-mail: {zhaoziming, fanzhang}@zju.edu.cn.*
- *Sheng Gao is with the School of Information, Central University of Finance and Economics, Beijing, 100081, China. E-mail: sgao@cufe.edu.cn.*

Security issues of smart contracts. Currently, smart contracts from various fields hold more than $10 billion worth of virtual currencies. This gives much incentive to malicious users for discovering and exploiting potential vulnerabilities in smart contracts. For example, in May 2019, Binance suffered a hacking attack that resulted in the theft of over 7,000 Bitcoins [6], and then it also lost $100 million of cryptocurrency due to the cross-chain bridge contract's vulnerability in October 2022. In addition to the economic value carried by smart contracts, these attacks stem from several critical characteristics. First, smart contracts run in a permission-less network, which means hackers can check them freely, and try to find their bugs. Second, the consensus protocol makes them immutable once deployed, requiring developers to anticipate all possible execution statuses, which is undoubtedly difficult. Therefore, effective vulnerability checkers[1] are essential to ensure smart contracts are bug-free and well-designed before deploying them to Ethereum.

Key challenges in contract vulnerability detection. The existing contract analysis arts leverage techniques such as pattern matching (*e.g.,* SmartCheck [7] and Slither [8]),

1. In order to enable developers or auditors to understand the contract status comprehensively, the vulnerabilities or defects mentioned in this paper not only refer to *contract bugs*, but also include *code optimizations*.

TABLE 1
Qualitative comparison of VulHunter and existing smart contract vulnerability detection methods.

Method Name	Required Input	Technology	Manual	Interpretable	Extensible	Path	Locate	Verify&Simulate	Speed
SmartCheck [7]	Source code	Pattern matching	rules	Yes	Medium	No	Yes	No	Medium
Slither [8]	Source code	Pattern matching	rules	Yes	Medium	No	Yes	No	Medium
NeuCheck [14]	Source code	Pattern matching	rules	Yes	Medium	No	Yes	No	Medium
Zeus [3]	Source code	Pattern matching	rules	Yes	Medium	No	Yes	No	Medium
SMARTEMBED [15]	Source code	Similarity code matching	No	Yes	Easy	No	Yes	No	Medium
TMP [12]	Source code	ML (GNN)	No	Yes	Easy	No	No	No	Medium
Peculiar [16]	Source code	ML (Pre-training)	No	Yes	Easy	No	No	No	Medium
DeeSCVHunter [17]	Source code	ML (Integration framework)	No	No	Easy	No	No	No	Fast
S-gram [13]	Source code	ML (N-gram)	No	No	Easy	No	No	No	Fast
Oyente [4]	Source code and bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Osiris [18]	Source code and bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Securify [19]	Source code and bytecode	Pattern matching	rules	Yes	Medium	Yes	Yes	No	Medium
Mythril [9]	Source code and bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
DefectChecker [20]	Source code and bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Fast
Maian [21]	Bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Eosafe [22]	Bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Honeybadger [23]	Bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Manticore [24]	Bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
teEther [25]	Bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Sailfish [26]	Bytecode	Symbolic execution	rules	Yes	Hard	Yes	Yes	Yes	Slow
Contractguard [27]	Bytecode+transaction	Path matching	Review	Yes	Medium	No	No	No	Medium
Contractfuzzer [10]	Bytecode+ABI	Fuzzy testing	Oracles	Yes	Hard	Yes	No	Yes	Slow
ILF [11]	Bytecode+transaction	Fuzzy testing	Oracles	Yes	Hard	Yes	No	Yes	Slow
SMARTIAN [28]	Bytecode	Fuzzy testing	Oracles	Yes	Hard	Yes	No	Yes	Slow
Contractembed [29]	Bytecode	ML (Graph embedding)	No	Yes	Easy	No	No	No	Medium
ContractWard [30]	Source code and bytecode	ML (XGBoost etc.)	No	No	Easy	No	No	No	Fast
VulHunter (Ours)	**Source code, bytecode and opcode**	**ML+Symbolic execution**	**No**	**Yes**	**Easy**	**Yes**	**Yes**	**Yes**	**Fast**

symbolic execution (*e.g.,* Oyente [4] and Mythril [9]), fuzzy testing (*e.g.,* Contractfuzzer [10] and ILF [11]), and machine learning (ML) (*e.g.,* DR-GCN [12] and S-gram [13]) to identify vulnerabilities, as described in Table 1. However, it is very challenging for them to conduct automated inspections that address the following challenges at the same time.

Challenge 1: (Requirement) Support analysis of source code and bytecode/opcode simultaneously. The source code of smart contracts is usually developed using a high-level programming language, such as Solidity [31]. It has been analyzed by many methods due to its legibility, such as pattern matching (*e.g.,* NeuCheck [14] and Zeus [3]), similarity code matching (*e.g.,* SMARTEMBED [15]), and machine learning (*e.g.,* Peculiar [16] and DeeSCVHunter [17]). Nonetheless, according to the latest records, among the 1 million smart contracts running on Ethereum, only less than 2% open their source code [18], [32], [33]. Also, contracts usually invoke others, and the callee contracts may not open their source code for inspection. Since these works can only build intermediate representations/graphs based on the source code, it is difficult to analyze the vast contracts on Ethereum.

Instead, the Ethereum Virtual Machine (EVM) bytecode is compiled from the contract source code and stored in each node on the Ethereum system. Everyone can check it and convert it from/to opcode (*i.e.,* the code executed directly on the EVM) unconditionally and lossless. Therefore, in order to meet the various developers' requirements, a practicable contract checker should work with the bytecode/opcode, not just the source code. However, this is hard to implement for the following reasons. (i) The bytecode loses some contract semantics. When compiling a smart contract to bytecode, EVM will refine the source code, which means some information will be removed or optimized, so it is hard to know the original semantics of the source code from the bytecode. For instance, detecting whether functions have return values in the source code is straightforward. However, this is difficult to complete at the bytecode level as the EVM will automatically add default values for functions without return values. Also, the more details are discussed in § 5.2.1.

(ii) The contract introduces a lot of benign interference. A vulnerability generally involves few statements, yet vulnerable contracts hold many vulnerability-irrelevant statements, called noise code, which may confuse code matching.

Challenge 2: (Intelligent & Unmanned) Reduce the reliance on pre-manually defined rules/patterns and expert involvement. Even though there have been some studies on vulnerability detection based on bytecode, such as symbolic execution (*e.g.,* teEther [25] and DefectChecker [20]) and fuzzy testing (*e.g.,* Contractfuzzer [10] and SMARTIAN [28]), there is still a growing need to detect and prevent more and more kinds of contract vulnerabilities. A main limitation of these methods is that they require specific vulnerability patterns/oracles or specification rules (collectively patterns) defined by experts to construct vulnerability detectors and/or code inspectors. This hinders their application to Ethereum for the following reasons. (i) The manually defined patterns are subject to the knowledge of expensive contract experts and bear the risk of errors. Also, some complex vulnerabilities are non-trivial to be covered completely. For example, it is difficult to describe consistency rules for the transaction order dependent (TOD) vulnerability manually, and define bytecode-level patterns to consider all expressions of vulnerabilities, such as reentrancy. Even crafty attackers may use tricks to bypass fixed patterns.

(ii) The diversity of bytecode generation will impede the formation of vulnerability patterns. Currently, there are dozens of compiler versions, and a compiler may generate different bytecode for the same code pieces under diverse versions. As mentioned in [29], the bytecode similarity between the newer and older compilers is only 77.8% for the same contract. Therefore, the bytecode-based patterns may be ineffective just as vulnerable/defective codes were compiled with different versions and contain distinct instructions.

(iii) With the race between attackers and defenders, it can be far too slow and costly to write new patterns/oracles in response to the emerging vulnerabilities created by attackers.

Challenge 3: (Practical) Help developers to complete the contract lifecycle safely, such as vulnerability identification, verification, repairment, simulation, and monitoring. To mitigate the above limitations, ML-based methods (*e.g.,* TMP [12] and ContractWard [30]) are used for automated learning of contract vulnerability features, thereby making full use of existing vulnerable (refers to defective and optimizable) contracts to express vulnerabilities more perfectly. Nevertheless, these approaches are limited in terms of scalability, generalizability, and interpretability, giving their insufficient detection accuracy and running speed. More importantly, as shown in Table 1, they only inspect whether the contract is vulnerable, which is not enough to help developers to fix vulnerabilities, let alone verification, simulation, and monitoring. Therefore, it is an emerging yet crucial issue to detect various vulnerabilities (*e.g.,* reentrancy and timestamp) of contracts in an effective, efficient, and interpretable manner, while enabling developers to finish the contract lifecycle more safely in the real world. Specifically, (i) report the defective source code statements, key bytecode fragments, and possible suggestions to them for further contract repairs during contract development and deployment phases. (ii) Support automated vulnerability verification to eliminate false positives and reduce the workload of manual review. (iii) Provide possible inputs for invoking contracts to trigger the vulnerabilities. (iv) Monitor contract calls to judge abnormal behavior in contract execution and destruction stages.

An insight into achieving these services is to combine ML with traditional methods instead of using the former alone, given the unique advantages of traditional methods, *e.g.,* symbolic execution-based arts can obtain the contract inputs triggering the vulnerabilities. This further necessitates ML-based methods to focus on the critical slices of contract runtime execution sequences in the bytecode/opcode form, rather than the entire contract bytecode. However, since a contract generally contains multiple incomprehensible execution sequences, it is challenging to get the specific labels of sequences when only knowing their contract vulnerability categories in the training dataset, so that the ML classifiers for identifying sequences cannot be trained. This problem is also known as the *classification lacking fine-grained labels*.

Contribution. To overcome the above challenges, we propose VulHunter, a method that can effectively detect vulnerable bytecode/opcode paths without manual pre-defined patterns. It extracted the contract execution sequences/paths based on the opcode, and completed the process of source code-to-bytecode-to-opcode conversion combined with the contract compilers such as Solc [34]. Then, it leverages multi-instance learning (MIL) to infer the fine-grained labels of contract execution sequences automatically, and employs a Bag-instance/self-model attentions based Bi-directional Long Short-Term Memory (Bi^2-LSTM) model to inspect them accurately and output the vulnerable sequences with attention vectors. Also, it extracts the key opcodes with large weights in sequences, and locates the defective contract source code statements by mapping from the assembly language source code file. Furthermore, based on symbolic execution technology, the reported vulnerable sequences are used to construct and solve the symbolic constraints to validate their feasibility. Meanwhile, the solved parameters can be obtained to trigger the vulnerabilities such as *integer-overflow*, and the abnormal contract calls can be determined by verifying their inputs with the constraints. This paper mainly presents the design, implementation and evaluation of VulHunter. In total, we make the following contributions:

- *Comprehensive design requirements.* We examine the numerous contract vulnerability types (*c.f.,* § 2) and security analysis arts (*c.f.,* § 6), and further clarify the field demands (*i.e.,* three challenges) to guide our designs. To the best of our knowledge, VulHunter is the most accurate and practical contract vulnerability method based on ML and symbolic execution.
- *Novel detection approach.* We design and develop VulHunter with six components, such as Vulnerability Learner, to detect vulnerabilities in contract source code or bytecode/opcode without expert involvement (*c.f.,* § 3), thereby meeting the challenges (i)-(ii). It employs Bi^2-LSTM model to identify/output the vulnerable runtime execution sequences and defective source code statements. Also, it delivers an optional constraint-solving module to construct the constraints of vulnerable sequences and compute contract inputs automatically, thus addressing the challenge (iii).
- *Superior analytical performance.* We evaluate the performance of VulHunter on five open source datasets (*c.f.,* § 4). Compared with SOTA methods based on various technologies such as pattern matching and symbolic execution, our solution can detect contract vulnerabilities more accurately, efficiently, and robustly. Also, VulHunter is flexible given it can be configured with various baseline models and hyperparameters to adapt to diverse user requirements. More importantly, compared with ML-based arts, it can produce classification results accurately while providing defective source code statements, key opcode fragments, and vulnerable runtime execution paths, benefiting the automated validation and vulnerability repair. Finally, we theoretically demonstrate its effectiveness and discuss the limitations, improvements, and more application scenarios in § 5.

2 Background and Motivation

In this section, we briefly introduce the background information about smart contracts, as well as their vulnerabilities.

2.1 Smart Contracts

Operation procedure or lifecycle. The procedure of smart contracts consists of four states: development, deployment, execution, and destruction. Users can develop a contract with solidity language and deploy it to the Ethereum platform. Then, the contract source code will be compiled to EVM bytecode and identified by a unique 160-bit hexadecimal hash called contract address. The contract holds an amount of virtual currency Ether (called balance), whose execution depends on its code. It usually runs on a permissionless network, and anyone can invoke its methods through ABI (Application Binary Interface) [31]. Specifically, the user at

TABLE 2
List of opcode instructions.

Type	Instructions
Compute	ADD, MUL. SUB, DIV, MOD, ADDMOD, EXP, SIGNEXTEND, ...
Compare	LT, GT, SLT, SGT, EQ, ISZERO, ...
Bit operation	AND, OR, XOR, NOT, BYTE, SHL, SHR, SAR, ...
Transaction data	ADDRESS, BALANCE, ORIGIN, CALLER, GAS, ...
Memory	MLOAD, MSTORE, MSTORE8, SLOAD, SSTORE, MSIZE, CREATE, DELEGATECALL, STATICCALL, CALL, ...
Call data & Code data	CALLCODE, CALLVALUE, CALLDATALOAD, CALLDATASIZE, CALLDATACOPY, CODESIZE, CODECOPY, EXTCODECOPY, ...
Block data	GASPRICE, GASLIMIT, DIFFICULTY, NUMBER, TIMESTAMP, COINBASE, BLOCKHASH, SHA3(KECCAK256), ...
Jump & Stop	STOP, JUMP, JUMPI, PC, GETPC, RETURN, REVERT, INVALID, SELFDESTRUCT, RETURNDATASIZE, RETURNDATACOPY, ...
Stack	PUSH1-32, POP, SWAP1-16, DUP1-16, LOG0-4, ...

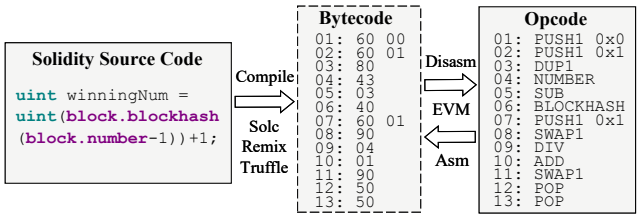


Fig. 1. The relationships among source code, bytecode and opcode.

address α_U can call the contract by sending a transaction $T = \langle \alpha_U, \alpha_A, E, G, D, \ldots \rangle$ to the contract address α_A, where E, D, and G denote the input amount, call parameters, and execution cost, respectively. Finally, the contract owner can destroy the contract by invoking the "selfdestruct" function.

Ethereum development language. Solidity is an object-oriented and Turing-complete programming language for implementing contracts on various blockchain platforms, most notably, Ethereum. Its grammar is similar to JavaScript, *e.g.*, they both implement object-oriented features such as inheritance and complex user-defined types. Nevertheless, Solidity has some unique properties. For example, it can provide keywords like payable to mark payment operations, thus making transfer operations easier. Moreover, although some alternative programming languages (*e.g.*, Obsidian [35] and Vyper [36]) have been proposed, Solidity is still the most popular language in Ethereum. Notably, the diversity of contract languages is challenging for auditors. Fortunately, contracts developed in various languages on Ethereum are executed in EVM bytecode, so that the contract security can better benefit from bytecode-oriented analysis methods.

Ethereum Virtual Machine (EVM) bytecode and opcode. EVM is a stack-based machine that maintains a stack of uint256s to hold local variables, function arguments, etc. When a transaction needs to be executed, EVM will split bytecode into bytes, each representing a unique instruction called opcode. There are 150+ opcodes by April 2022 [37]. As described in Table 2, they can be divided into 9 types according to their function.[2] Monitoring them can benefit vulnerability detection. For instance, the opcodes of computing type (*e.g.*, ADD and MUL) perform arithmetic operations and can be used to identify *integer-overflow* vulnerabilities.

Furthermore, EVM utilizes these opcodes to execute the task. Fig. 1 shows an example of contract transformation

2. The opcodes are detailed in https://github.com/ContractAudit/VulHunter/tree/main/Opcodes.

Utilization \ Risk	High	Medium	Low	Informational	Optimization
Exactly	High	Medium	Low	Informational	Optimization
Probably	High	Medium	Low	Informational	Optimization
Possibly	Medium	Low	Low	Informational	Optimization

Fig. 2. Vulnerability assessment mechanism.

to illustrate the relationships among source code, bytecode, and opcode. The source code was compiled into bytecode 0x600060...5050 using compilers such as solc [34]. EVM splits this bytecode into bytes (0x60, 0x00, ..., 0x50), and executes the first byte 0x60, which refers to opcode PUSH1. PUSH1 pushes one-byte data (*i.e.*, 0x00) to the EVM stack. Then, EVM reads 0x60 and pushes 0x01 (*i.e.*, number 1) into the stack. Subsequently, it executes the remaining bytecodes, such as 0x80, 0x43, and 0x03, *i.e.*, opcodes DUP1, NUMBER, and SUB. Among them, NUMBER extracts the block number. SUB obtains the top two values from the stack, *i.e.*, block.number and 0x01, and puts their subtraction result into the stack.

2.2 Vulnerabilities in Smart Contracts

2.2.1 Definition of Impact Levels

A sound and reasonable vulnerability assessment scheme for contract bugs or code optimizations can help developers to understand their contract security better. To this end, combined with CVSS2.0 (Common Vulnerability Scoring System), the vulnerability severity of smart contract can be rated as *High*, *Medium*, *Low*, *Informational* (*Info*), and *Optimization* (*Opt*) in terms of risk degrees and utilization difficulties. The detailed partitioning is shown in Fig. 2. The risk degree refers to the impact of vulnerability on the blockchain systems, users, and other resources. According to the three impact dimensions of confidentiality (C), integrity (I), and availability (A), the harm degree is divided into *High*, *Medium*, *Low*, *Info* and *Opt*. The utilization difficulty refers to the possibility of vulnerability occurrence. Based on the attack cost (*e.g.*, money, time and technology), utilization condition (*i.e.*, the difficulty of attack utilization) and trigger probability (*e.g.*, vulnerabilities can only be triggered by a few people), it is ranked into *exactly*, *probably*, and *possibly*.[3]

2.2.2 Examples of Smart Contract Vulnerabilities

In order to understand the contract vulnerability detection at the bytecode/opcode level, we combine the contract code to explain some simple examples of vulnerabilities (*c.f.*, Table 3) supported by VulHunter in terms of occurrence principle, severity, repair strategies, and insights in bytecode.[4]

(i) Reentrancy with Ether (*reentrancy-eth*): Reentrancy vulnerability is a classic problem, which leads to the loss of assets with a market value of nearly $60 million in 2016 [10]. It refers to reentry with the following features: reentrant calling, Ether sending, and reading before writing.

Severity: High severity. RE can cause massive assets to be overspent or stolen (*High* risk). Also, it requires some

3. The vulnerability assessment scheme is detailed in https://github.com/ContractAudit/VulHunter/tree/main/Severity_assessment.

4. More vulnerabilities are illustrated in https://github.com/ContractAudit/VulHunter/tree/main/Vulnerability_examples.

TABLE 3
The severity of vulnerabilities (bugs and Opts) supported by VulHunter.

ID	Vulnerability Name	Severity	ID	Vulnerability Name	Severity
RE	reentrancy-eth	High	E20IF	erc20-interface	Medium
CAL	controlled-array-length	High	CTL	costly-loop	Low
SU	suicidal	High	TS	timestamp	Low
CDC	controlled-delegatecall	High	BP	block-other-parameters	Low
AS	arbitrary-send	High	CLL	calls-loop	Low
TOD	TOD	High	LLC	low-level-calls	Info
UIS	uninitialized-state	High	E20ID	erc20-indexed	Info
IE	incorrect-equality	Medium	E20TR	erc20-throw	Info
IO	integer-overflow	Medium	HC	hardcoded	Info
UCL	unchecked-lowlevel	Medium	AIB	array-instead-bytes	Opt
TO	tx-origin	Medium	UUS	unused-state	Opt
LE	locked-ether	Medium	COL	costly-operations-loop	Opt
UCS	unchecked-send	Medium	ST	send-transfer	Opt
BC	boolean-cst	Medium	BE	boolean-equal	Opt
E721IF	erc721-interface	Medium	EF	external-function	Opt

conditions to trigger (*probably* utilization). For instance, the auxiliary contract is needed to complete the attack.

```
contract PullPayment {
  mapping (address => uint) userBalances;
  function withdraw(){
    //Reenter the function
    if(!(msg.sender.call.value(userBalance[msg.sender])
         ())){ throw; }
    userBalance[msg.sender] = 0;
  }
}
contract Attack {
  PullPayment object;
  function attack() payable {object.withdraw(1 ether);}
  function() public payable {object.withdraw(1 ether);}
}
```

Listing 1. The sample of *reentrancy-eth*.

Example: An attack scenario is depicted in Listing 1. Bob constructs an *Attack* contract and performs the "withdraw()" function by invoking the "attack()" function, which will trigger the fallback function. By this means, Bob implements multiple calls to "withdraw()". Since the "userBalance" variable hasn't changed before the secondary call , Bob obtained more than the amount he deposited into the contract.

Improvements to contracts: Put userBalance[msg.sender] = 0 before the call function. That is, the contracts should use the check-effects-interactions pattern to avoid this vulnerability.

Possible insight at bytecode-level: Ethereum provides three methods to transfer Ethers, *i.e.,* address.send(), address.transfer(), and address.call().value(). These methods all generate a CALL instruction, which reads seven values from the EVM stack. The first three values represent the gas limitation, recipient address, and transfer amounts, respectively. The CALL instruction that meets the following conditions is almost generated by call().value(). (i) the gas limitation does not contain a specific value "2300"; (ii) the transfer amount is greater than 0. In addition, the SLOAD instruction is used to get a key value (named *Slot ID*) from the EVM stack and puts the mapping result read from storage back onto the stack [37]. If the conditional expression contains the SLOAD instruction and its *Slot ID* are written by the SSTORE instruction after executing the CALL instruction, it means the CALL instruction can be executed again and cause Reentrancy. Among them, the SSTORE instruction is used to save data into storage, and it reads two values from the EVM stack, *i.e., Slot ID* and the stored value.

(ii) Transfer replaces send (*send-transfer*): Both the send and transfer functions specify that the operation has a limit of 2300 gas, but a failure of the send function does not trigger the exception and can be reentered easily. Thus, developers are recommended to use the transfer function.

Severity: Opt severity. ST can help developers improve the security of transfer operations (*Opt* risk). Also, it can be triggered by executing vulnerable codes (*exactly* utilization).

Example: As shown in Listing 2, the "func" function utilizes "send" to extract amounts, which can be improved.

Improvements to contracts: It is suggested to replace "send" with the "transfer" to keep the funds secure.

Possible insight at bytecode-level: There exists a CALL instruction that meets (i) the gas limitation contains a specific value "2300"; (ii) the transfer amount is larger than 0.

```
contract Crowdsale {
  address owner = msg.sender;
  modifier verify() { require(tx.origin == owner); _; }
  function func(address payable dst) payable verify(){
    dst.send(msg.value);
  }
}
```

Listing 2. The sample of *send-transfer*.

3 The VulHunter Approach

In this section, we elaborate on the design principles and workflow of VulHunter, as well as its components.

3.1 Design Overview

Fig. 3 depicts an overview architecture of VulHunter. VulHunter can take the Solidity source code, bytecode, or opcode of smart contracts as input, and eventually output contract vulnerabilities (*e.g.,* RE, TO, and TS) with their severity as described in § 2.2, as well as the corresponding vulnerable runtime opcodes and their key fragments. In particular, it can highlight the defective contract statements when analyzing the source code. Also, the runtime opcodes can be used to build symbolic constraints and then execute secondary verification and utilization. Specifically, VulHunter contains six components. Contract Inputter is responsible for generating the contract opcodes, and CFG Builder constructs a CFG with three kinds of blocks. Then, Instance Builder performs the depth-first traversal to obtain the runtime opcode sequences (called instances). Vulnerability Learner captures vulnerability features automatically by training detectors on benign and malicious (vulnerable) contracts. Vulnerability Identifier employs the detectors to identify the vulnerable instances of contracts. Finally, Result Exporter locates the defective source code statements, validates the instance feasibility, and outputs the contract audit reports.

3.2 Contract Inputter

As shown in the left part of Fig. 3, Contract Inputter can feed the source code, bytecode, and opcode of contracts as input. Specifically, due to the opcode is directly used by CFG builder, the source code needs to be compiled into the bytecode, and then disassembled into opcodes using the API of Geth [38]. Fig. 1 depicts this process vividly. In particular, the compiler can resolve multiple associated contracts uniformly to generate their bytecode. Also, the assembly language source code file (ASM) can be obtained during the contract compiling and further used to map bytecodes/opcodes to source codes, as illustrated in § 3.7.

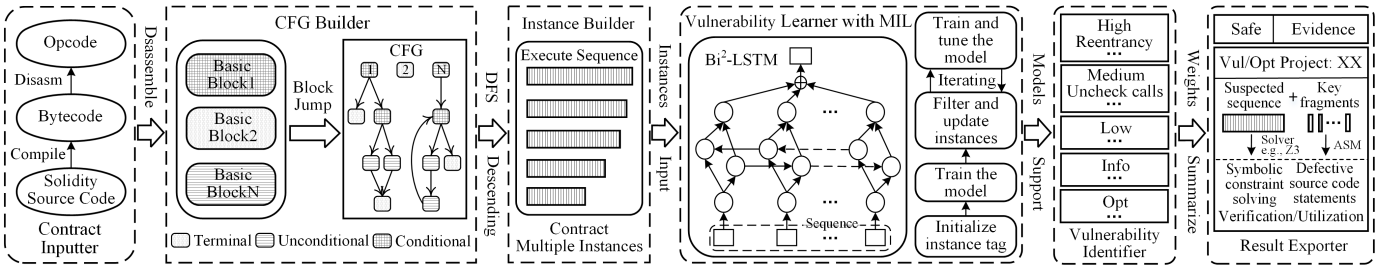


Fig. 3. Overview architecture of VulHunter.

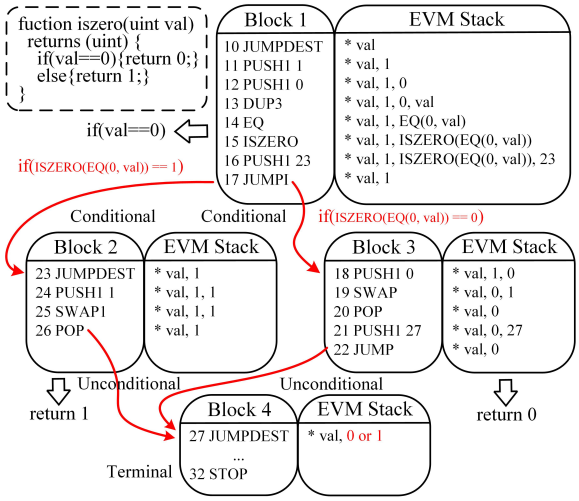


Fig. 4. Example of symbolic reasoning by the CFG Builder.

3.3 CFG Builder

CFG Builder constructs the CFG of contracts based on their opcodes to explore the state transitions during the actual execution. This process is similar to the methods such as DefectChecker [20] and EtherSolve [39], including block identification and edge inference. At first, he splits the opcode into several basic blocks. A basic block is a straight-line code sequence without branches except for the entry and the exit. Its type is presented by the exit instruction, which usually marks the end of the continuous operations. If the last instruction is JUMPI, the block type is conditional. If the last instruction belongs to the stop type shown in Table 2 (*e.g.,* STOP and RETURN), the block type is terminal. The blocks that do not fall into both types are assigned as unconditional.

Then, CFG Builder performs symbolic inference on the instructions in each block to establish connections with neighboring blocks. Different from other stack-based machines (*e.g.,* JVM), the jump positions of EVM opcodes need to be computed during instruction reasoning. Specifically, when operating an instruction, it reads several symbolic states from the top of the EVM stack and puts the computation result back on the stack. Through continuous reasoning operations, we can obtain jump relations between blocks, and their types are consistent with blocks, *i.e.,* conditional and unconditional.

As an example shown in Fig. 4, there are 4 blocks, each containing several instructions. The instructions in block 1 represent the code if(val==0). Block 2 and block 3 put the value (1 or 0) to the EVM stack, respectively. Block 4 returns the value (0 or 1) to the environment. The leftmost number in each line indicates the instruction index ID, and the middle part is the instruction that needs to be reasoned. All instructions will reason sequentially based on their index ID. There is a Program Counter (PC) that records the ID executed at the current time. Specifically, the PC starts from ID 10 in block 1. Before the EVM executes the JUMPDEST instruction, there is a symbol "val" in the EVM stack, which represents the input value of the "iszero" function. JUMPDEST marks a valid jump destination and does not read or push any values. Then the PC points to ID 11, and EVM pushes a value 1 to the EVM stack. Also, "0" is pushed into the EVM stack and PC point to 13. DUP3 duplicates the 3rd stack item, *i.e.,* the symbol "val" is pushed into the EVM stack again. EQ reads two values from the EVM stack. If the two values are equal, then the EVM pushes 1 into the stack, otherwise 0 is pushed. After that, ISZERO reads a value from the top of the EVM stack. If the value is equal to 0, then 1 is pushed into the stack, otherwise 0 is pushed. JUMPI (ID 17) reads two values from the stack, *i.e.,* the jump position and a conditional expression. According to the expression's result, the PC will conditionally jump to the positions of IDs 23 and 18, respectively.

When the PC points to ID 23, it will execute the instructions on IDs 23-26, otherwise 18-22, and both eventually jump to block 4 unconditionally. When performing the first instruction of block 4, the EVM stack holds two values, *i.e.,* val and 0/1. Eventually, block 4 (type of terminal) returns the value 0/1 and uses the STOP instruction to finish the execution. Notably, given the complex computation of destinations for jumps without immediate target offsets, the sensitive observations of ML models such as Bi^2-LSTM (*c.f.,* § 3.5) can tolerate slight CFG biases, thus enabling VulHunter to deliver robust and accurate detection[5]. Also, the feasibility validation (*c.f.,* § 3.7) of paths may mitigate the impacts of imprecise/unsound CFGs, and multi-threaded operations can be used to improve the efficiency of CFG construction.

3.4 Instance Builder

In order to make VulHunter discover vulnerable execution paths, the Instance Builder focuses on each opcode sequence (called instance) that the contract actually runs, rather than a hodgepodge of all opcodes. The process of instance extraction is detailed in Algorithm 1. Specifically,

5. The CFG building process and its correctness analysis are detailed in https://github.com/ContractAudit/VulHunter/tree/main/CFG.

Algorithm 1 Opcode Execution Sequence Extraction

Input: Contract source code / bytecode / opcode; maximum number of block cycles n_{cycle}; maximum length of blocks n_{block}; number of sequences n_{seq}
Output: The set of opcode execution sequences $opseqs$

1: *Recursively traverse blocks to obtain the opcode sequences*
2: **procedure** SEQDFS(*seq*, *names*, *block*, *num*, *seqset*)
3: *num* += 1
4: **if** *block*.name $\notin$ *names* **then** *num* = 0
5: *names* $\cup =$ *block*.name, *seq* $\cup =$ *block*.opcodes
6: **if** *num* == n_{cycle} or len($names$) $\geq n_{\text{block}}$ or
7: *block*.type == terminal **then** *seqset* $\cup =$ *seq*;
8: **else for** $block_{\text{next}} \in$ *block*.outblocks **do**
9: *seqset* = SEQDFS(seq,$names$,$block_{\text{next}}$,num,$seqset$)
10: **return** *seqset* ▷ Return the opcode sequences
11: *Step 1: Build CFG of the contract*
12: *source code* $\xrightarrow{\text{compile}}$ *bytecode* $\xrightarrow{\text{disasm}}$ *opcode*
13: *blocks* = **block_split**(*opcode*)
14: *cfg* = **construct**(*blocks*) ▷ Build the links between blocks
15: *Step 2: Obtain the execution sequences of the contract*
16: *opseqs* = $\emptyset$ ▷ Initialize the set of opcode sequences
17: **for** *block* $\in$ *cfg*.blocks **do**
18: **if** *block*.inblocks == $\emptyset$ **then**
19: *opseqs* $\cup =$ SEQDFS($\emptyset$, $\emptyset$, *block*, 0, $\emptyset$)
20: *opseqs* = sorted(*opseqs*,key=lambda d:len(d),reverse=True)
21: **return choose**(*opseqs*, n_{seq}) ▷ Choose the opcode sequences

CFG Builder constructs the contract CFG through the steps in § 3.3 (lines 12-14). Then, Instance Builder obtains the instances by performing the procedure SEQDFS (lines 2-10) from the initial blocks according to the CFG (lines 16-19). Furthermore, SEQDFS leverages the depth-first traversal to record in-block opcodes along the execution path. The search for the execution path is stopped when it meets one of the three conditions. (i) Continuous n_{cycle} blocks that have been searched, which is considered as a cyclic execution path. (ii) The length of the path exceeds the limit n_{block}. (iii) The type of the last block is terminal, representing the ends of paths.

Finally, due to sequence space explosion and performance limitations, the Instance Builder outputs the contract's n_{seq} selected instances (lines 20-21). Note that he has various options such as random selection and ordered assignment. In § 4.7, we tested the performance of different selection schemes (*e.g.,* longest, shortest and interval) and the number of selected instances under the same other settings, *i.e.*, control variables. The instances with longer lengths generally achieve better effects, which may be attributed to the fact that they hold more opcodes and semantic information.

3.5 Vulnerability Learner

To detect contract vulnerabilities at the bytecode level, Vulnerability Learner leverages multi-instance learning mechanism to identify malicious contract behaviors automatically from the instances extracted by Instance Builder. Specifically, it trains a binary classification model for each vulnerability, and multiple models grant VulHunter the ability to analyze contracts comprehensively. In this part, taking a model as an example, we introduce the process of vulnerability feature learning and vulnerable instance detection, which consists of three parts: instance label initialization, model training/classification, and instance optimization, as depicted in Fig. 5. Also, its effectiveness/correctness will be discussed in § 5.1.2.

(i) Instance tag initialization. During the vulnerability feature learning stage, the training dataset includes instances *opseqs* of multiple benign and malicious contracts, and each contract can be vividly described as a bag. However, since it is challenging to understand the opcodes and label each instance of contracts, we only determine whether the contract is malicious/vulnerable (*i.e.,* existing the target vulnerability) and cannot get the specific tags of its instances. In other words, malicious contracts include at least one vulnerable instance, yet the specific instance tag is unknown. This is known as the problem of missing fine-grained labels, preventing the model from being trained on instances. To this end, we first initialize the instance tag to coincide with its bag and integrate all instances as the training dataset $\mathcal{D}_0$.

(ii) Model training and classification. To distinguish between benign and malicious instances, we employ a basic Bi-directional LSTM (Bi-LSTM) model to focus on the underlying contextual relationships of the opcodes in instances. Next, the Bag-instance and self-model attentions based Bi-LSTM (called Bi^2-LSTM) model is proposed to catch the salient instance fragments and consider both bag and instance learning effects. Notably, given the generality and extensibility framework of Vulnerability Learner, he can also employ other ML models (*e.g.,* Random Forest) to further improve the detection performance, which is discussed in § 4.6. This part details the components of Bi^2-LSTM model.

Instance encoding. The instance consists of T opcodes $opseq = \{x_1, \cdots, x_T\}$, and each opcode x_i is converted into its bytecode e_i by assembly. Then the vector is fed into the next layer as a real-valued vector $\mathcal{C}_{opseq} = \{e_1, \cdots, e_T\}$.

Bi-LSTM network. For the opcode sequence modelling task, it is beneficial to consider future and past contexts. To this end, Bi-LSTM networks extend the unidirectional LSTM networks by introducing a second layer, where the hidden to hidden connections flow in the opposite temporal order. As shown in step 2 of Fig. 5, the Bi^2-LSTM layer contains two sub-networks of LSTM units for the left and right sequence context, representing the forward and backward passes, respectively. In the forward-pass t^{th} time step operation, the forget gate f_t and input gate v_t can be calculated as

$$f_t = \sigma(W_f \cdot [\overrightarrow{h_{t-1}}, e_t] + b_f) \quad (1)$$

$$v_t = \sigma(W_v \cdot [\overrightarrow{h_{t-1}}, e_t] + b_v) \quad (2)$$

where $\overrightarrow{h_{t-1}}$ is the current hidden state and e_t denotes the t^{th} input of the LSTM unit. Then the temporary memory cell $\tilde{C}_t$ and the next cell state can be computed and updated by

$$\tilde{C}_t = \tanh(W_c \cdot [\overrightarrow{h_{t-1}}, e_t] + b_c) \quad (3)$$

$$\overrightarrow{C_t} = f_t \cdot \overrightarrow{C_{t-1}} + v_t \cdot \tilde{C}_t \quad (4)$$

Finally, the output gate o_t and the next hidden state $\overrightarrow{h_t}$ can be obtained as follows. Particularly, W (*i.e.,* W_f, W_v, W_c, and W_o) and b (*i.e.,* b_f, b_v, b_c, and b_o) are learnable parameters.

$$o_t = \sigma(W_o \cdot [\overrightarrow{h_{t-1}}, e_t] + b_o) \quad (5)$$

$$\overrightarrow{h_t} = o_t \cdot \tanh(\overrightarrow{C_t}) \quad (6)$$

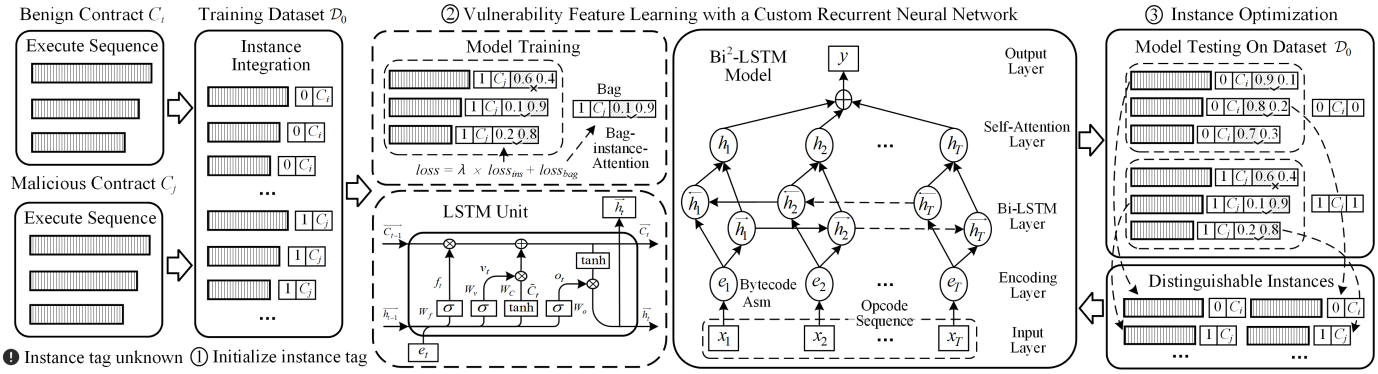


Fig. 5. The architecture of Vulnerability Learner with multiple instance learning.

The data flow of the backward-pass operations is similar to the above process, and the final output h_t of the Bi-LSTM layer for the t^{th} input is shown in the following equation:

$$h_t = \left[\overrightarrow{h_t} \oplus \overleftarrow{h_t}\right] \tag{7}$$

Self-model attention. Let H be a matrix consisting of output vectors $[h_1, h_2, \cdots, h_T]$ that the Bi-LSTM layer produced, where T is the sequence length. The sequence's representation r is formed by a weighted sum of these output vectors.

$$\tilde{H} = \tanh(H) \tag{8}$$

$$\alpha = \text{softmax}(w^* \tilde{H}) = [\alpha_1, \alpha_2, \cdots, \alpha_T] \tag{9}$$

$$r = H\alpha^* \tag{10}$$

where $H \in \mathbb{R}^T$, w is a trained parameter vector, and w^* is its transpose. The dimensions of w, α, and r are 1, T, and 1, respectively. Notably, the weight α reflects the importance of the input x_t in each time step t during model inference and can be utilized to calculate the key sequence fragments, as further detailed in § 3.7. Therefore, this mechanism enables VulHunter to discover defective contract fragments. Then, the final sequence-pair representation $h^* = \tanh(r)$ is obtained.

Classifying. We use a softmax function to predict the label $\hat{y}$ from a discrete class set $Y = \{0, 1\}$ for a sequence *opseq*.

$$\hat{p}(y|opseq) = \text{softmax}\left(W^{(opseq)} h^* + b^{(opseq)}\right) \tag{11}$$

$$\hat{y} = \arg\max_{y \in Y} \hat{p}(y|opseq) \tag{12}$$

Training with Bag-instance hybrid attention. As shown in the model training part of Fig. 5, Vulnerability Learner utilizes Bi^2-LSTM to classify each instance of bags and takes the cross entropy as the loss $loss_{ins}$ of the instance features learning. Specifically, the average $loss_{ins}$ of the bag C_j can be calculated from Eq. 13, where $p(x)$ represents the probability that an instance x with the tag $y(x)$ (0 and 1 denoting benign and malicious, respectively) is predicted as malicious.

$$loss_{ins} = -\sum_{x \in C_j} \frac{(y(x)\log p(x) + (1 - y(x))\log(1 - p(x)))}{|C_j|} \tag{13}$$

More importantly, a key judgment is whether the bag is malicious, *i.e.,* there are malicious instances in the bag. To this end, Vulnerability Learner takes the result of predicting the most like the malicious instance (*i.e.,* instance with the largest $p(x)$) in the bag as the prediction result of the bag, and further calculates the cross-entropy loss $loss_{bag}$ combined with the tag y_{bag} of the bag C_j. Then, two losses are fused to constitute the Bag-instance hybrid attention mechanism, enabling the model to identify malicious instances in the bag guided by the bag/contract identification. This mechanism is the key to mitigating the impact of inevitable false instance labels during the training process, as detailed in § 5.1.1.

$$p_{bag} = \max([p(x)]_{x \in C_j}) \tag{14}$$

$$loss_{bag} = -y_{bag} \log p_{bag} + (y_{bag} - 1)\log(1 - p_{bag}) \tag{15}$$

$$loss = \lambda loss_{ins} + loss_{bag} \tag{16}$$

where a larger value of the weight $\lambda \in (0, 1] \in \mathbb{Q}^+$ indicates that the learner concentrates on instances, otherwise bags.

(iii) Instance optimization. As shown in step 3 of Fig. 5, the training dataset was updated after the Bi^2-LSTM training, and then used for iterative training again. Specifically, the Vulnerability Learner employs the trained model to identify the original datasets $\mathcal{D}_0$, and filters some distinguishable instances to construct a new training dataset. The dataset consists of two parts, one is from the benign bags predicting like the benign instances, *i.e.,* for the bag C_i, the instances are arranged in ascending order of $p(x)$, q_b of the selected instances $[sorted(C_i)[k]]_{k \in \{1, \cdots, \lceil q_b |C_i| \rceil\}}$ has a smaller $p(x)$. Another comes from the malicious bags predicting like the malicious instances, *i.e.,* for the sorted package C_j, and q_m of the selected instances $[sorted(C_j)[k]]_{k \in \{\lceil (1-q_m)|C_j| \rceil + 1, \cdots, |C_j|\}}$ has a larger $p(x)$. The value of q_m enables the model to consider vulnerabilities triggered by multiple instances, such as extracting contract permissions and stealing the balance. The impact of these parameters and training *epochs* on feature learning were evaluated in § 4.7, and their values were suggested by considering detection accuracy and overhead.

3.6 Vulnerability Identifier

Vulnerability Identifier employs multiple Bi^2-LSTM models to detect contract vulnerabilities. Specifically, for each vulnerability described in § 2.2, a detector that identifies contract instances with corresponding vulnerability features can be trained on pre-collected datasets. Note that the instance extraction of contract samples takes only once during the vulnerability detection. Then the model performs fast inference, so that multiple detectors only require little time overhead

and outperform many SOTA methods, which is discussed in § 4.4. Also, since different models are independent during inference, technologies such as parallel computing [40] can be used to improve performance. More importantly, given the scalability of Vulnerability Learner, detectors can be easily trained based on datasets to identify new vulnerabilities.

3.7 Result Exporter

After the vulnerability detection, the Result Exporter generates a security analysis report for the contract, which consists of security conclusions and repair suggestions.[6] Specifically, he outputs the "safety" conclusion and corresponding evidence (*i.e.,* analysis details) when there is no instance with vulnerability features. Otherwise, the contract vulnerabilities with their severity (*i.e., High*, *Medium*, *Low*, *Info*, and *Opt*) are indicated. More importantly, the vulnerable instances and their key fragments can be further used to perform symbolic constrant solving and defective source codes mapping, thus granting VulHunter the ability to execute secondary verification and utilization. Also, he can output the possible suggestions based on the above information to enable developers to fix the vulnerabilities.

Locating the defective source code statements. After the model prediction for contracts, Result Exporter obtains the contracts' instances with labels and their weight vectors α from Vulnerability Identifier. He computes the index vector $\mathcal{L}$ of the largest m weight values and extracts the corresponding opcodes ops_{key}, as stated in Eqs. 17~18. When the contract source codes are provided, he can map the begin and end positions (*i.e.,* lines and columns) of these opcodes in the source codes from the ASM file extracted by Contract Inputter. Then, the defective contract statements are obtained by intercepting the source codes of the specific positions.

$$\sum_{t=1}^{T} \alpha_t = 1, \mathcal{L} = \arg \max_{t \in [1,T]} (\alpha, m) = [\ell_1, \ell_2, \cdots, \ell_m] \quad (17)$$

$$ops_{\text{key}} = [x_t]_{t \in \mathcal{L}},\ begin, end \leftarrow \text{ASM}\,(ops_{\text{key}}) \quad (18)$$

Symbolic constraint solving for instances. The identified vulnerable instances describe the state transitions during contract execution, as described in § 3.3. Inspired by the symbolic execution technique, Result Exporter presents an optional constraint-solving module. It employs these instances to build the accumulating constraints satisfied by the symbolic inputs, which can be further translated to the Satisfiability Modulo Theories Library (SMT-LIB) language. This process is mature and performed in tools like Oyente [4] and Manticore [24]. Then, based on an SMT solver [41], it reports the instance is infeasible when its constraint condition is unsatisfiable. Otherwise, outputting the feasible conclusion and the input values that meet the conditions. As stated in [20], there will be some dead code in bytecode. In this way, the infeasible instances will be corrected as benign, thus eliminating some false positives. Currently, this module supports several SMT solvers (including Z3 [42], Yices [43], and CVC4 [44]), and users can employ either of them or a combination [45], given their unique performance and built-in theories. For instance, Z3 can handle linear/non-linear operations on more data types such as bitvectors and arrays, while Yices can achieve a faster solving process [46]. Notably, compared with symbolic execution-based methods, VulHunter can be regarded as an automatic rule-maker for vulnerabilities and a filter for contract execution paths (works like reinforcement learning [11]), which performs fast reasoning to prune normal paths. Furthermore, other feasible and misreported instances are discussed in § 5.3.1.

Secondary verification and utilization. In addition to allowing developers to determine the audit correctness quickly through instance feasibility detection and defective statement positioning, VulHunter can also support other services not provided by current ML-based methods. For instance, auditors can invoke the vulnerable contract with the parameters calculated by symbolic constraints to trigger the vulnerabilities such as *integer-overflow*. Also, they can identify abnormal contract transactions by verifying vulnerability constraints, and these applications are detailed in § 5.4.

4 Evaluation

In this section, we comprehensively evaluate the performance of VulHunter on open-source datasets and real-world contracts. Section 4.1 describes the baselines and related settings. Sections 4.2~4.9 answer the following research questions:

- **RQ1.** [Effectiveness] What is the effectiveness of VulHunter compared to SOTA methods in detecting contract vulnerabilities based on source code and bytecode?
- **RQ2.** [Production] How many vulnerabilities are present in the Ethereum blockchain?
- **RQ3.** [Performance] How much overhead (such as time and memory) does VulHunter require to analyze contracts?
- **RQ4.** [Authenticity] Can VulHunter discover contracts with substantial and serious vulnerabilities in Ethereum? The goal is to verify its effectiveness in real scenarios.
- **RQ5.** [Compatibility & Scalability] Can VulHunter support other baseline models for detection?
- **RQ6.** [Regularity] What is the effect of the variable parameters in VulHunter on its detection performance?
- **RQ7.** [Versatility] What are the advantages of VulHunter over other ML-based methods in vulnerability repair?
- **RQ8.** [Verification] What is the performance or capability of the constraint-solving module in VulHunter?

4.1 Experiments Setup

4.1.1 Baselines

We compare VulHunter with the latest version of the 9 SOTA methods (described in § 6) at the time of writing. As shown in Table 1, these methods can be divided into 4 classes according to their techniques, namely, pattern matching (Slither v0.8.3 [8], SmartCheck v2.0.3 [7], and Securify v1.0 [19] at GitHub commit 51ba124), symbolic execution (Oyente v0.2.7 [4], Mythril v0.22.41 [9], and DefectChecker [20] at GitHub commit e24c4c3), fuzzy testing (SMARTIAN [28] at GitHub commit 4543032) and ML (ContractWard [30] and TMP [12] at GitHub commit ab93541). In this section, we tested the 30 kinds of critical vulnerabilities mentioned in Table 3. As shown in Table 4, we manually annotated each vulnerability category detected by the methods with uniform labels.[7]

6. The examples of reports are shown in https://github.com/ContractAudit/VulHunter/tree/main/Reports_examples.

7. The mapping of vulnerabilities is detailed in https://github.com/ContractAudit/VulHunter/tree/main/VulnerabilityMapping.

TABLE 4
The kinds of vulnerabilities detected by the methods. Among them, the column "others" represents the number of vulnerabilities with each severity outside Table 2. Moreover, "∼" indicates that the vulnerability kinds can be extended easily.

Methods	High	Others	Medium	Others	Low	Others	Info	Others	Opt	Others
SmartCheck [7]	-	3	IE,UCL,TO,LE	10	CTL,TS,CLL	2	LLC,E20ID,E20TR,HC	12	AIB,ST,EF	3
Slither [8]	RE,CAL,SU, CDC,AS,UIS	11	IE,UCL,TO,LE,UCS, BC,E721IF,E20IF	12	TS,CLL	12	LLC,E20ID	11	UUS,COL, BE,EF	3
Securify [19]	RE,TOD	2	UCL,LE,UCS	1	-	1	-	1	-	0
Oyente [4]	RE,TOD	1	IO	3	TS	0	-	0	-	0
Mythril [9]	RE,SU,CDC	1	IO,UCL,TO,UCS	2	BP	2	-	0	-	0
DefectChecker [20]	RE	0	IE,UCL,TO,LE,UCS	0	CTL,BP,CLL	0	-	0	-	0
SMARTIAN [28]	RE,SU,CDC,AS	0	IO,UCL,TO,LE,UCS	2	TS,BP,CLL	1	-	0	-	0
TMP [12] ContractWard [30] **VulHunter (Ours)**	RE,CAL,SU, CDC,AS, TOD,UIS	∼	IE,IO,UCL,TO,LE, UCS,BC, E721IF,E20IF	∼	CTL,TS, BP,CLL	∼	LLC,E20ID, E20TR,HC	∼	AIB,UUS, COL,ST, BE,EF	∼

4.1.2 Implementation details

All experiments were performed on a PC running Ubuntu 18.04 and equipped with an Intel Core i7-10875H and 8GB of RAM. VulHunter is mainly implemented in Python with an estimated 15K lines of code.[8] Also, Bi^2-LSTM network is implemented with PyTorch, Adam optimizer [47], and CrossEntropyLoss function [48]. Besides, we employ solc with multiple Solidity versions such as 0.4.24 to compile source code into bytecode, utilize pyevmasm 0.2.3 to disassemble bytecode into opcode, and then use evm_cfg_builder 0.3.1 to build the CFG of contracts. Nowadays, the constraint-solving module is equipped with the Python versions of SMT libraries, including Z3 v4.12.1.0, Yices v2.6.4, and CVC4 v1.7.

4.1.3 Datasets

Table 5 shows the details of five datasets in the experiment. Datasets_1~2 are both open-source datasets with partial labels, which are suitable for assessing the precision of the methods (exploring RQ1, RQ5, and RQ6). In order to ensure the correctness of the labels, we have manually checked and supplemented these labels based on the verification results of muti-methods such as SmartFast [6], Slither [8] and Oyente [4]. We make these datasets and labels public to enable cross-checks from other researchers and further guarantee their accuracy. Among them, Dataset_1 consists of 38,600 real-world Ethereum contracts with source code, excluding the empty and uncompiled contracts. It has been used in [5], [12], [17], [49]. According to the vulnerability severity mentioned in Table 3, we can count the number of 30 kinds of vulnerabilities in Dataset_1 as *High* (13,149), *INFO* (19,659), *OPT* (38,314), etc.[9] Dataset_2 contains 579 contracts with only bytecode and is marked with 8 kinds of known vulnerabilities, which is the same used in [20]. In order to have a representative picture of vulnerabilities in the production environment, we have downloaded 13,413 contract Solidity codes in actual use by invoking the Etherscan API [32]. These contracts make up Dataset_3 (the size is 284.3MB). Also, we collected 183,710 contracts with runtime bytecodes as the Dataset_4, which were crawled and filtered from Ethereum by Chen et al. [50]. Note that the amount of bytecode far exceeds the source code, reflecting the necessity of bytecode-level analysis. Both datasets are employed to discuss the number of vulnerabilities in the Ethereum blockchain (exploring RQ2 ~RQ4 and RQ7). In addition, Dataset_5 consists of the collected contract source code for 29 well-known vulnerability events, which was used to further evaluate the authenticity of VulHunter.

8. VulHunter is available at https://github.com/ContractAudit/VulHunter/tree/main/VulHunter.

TABLE 5
Details of datasets.

DATASET	NUMBER OF ISSUES					CONTRACT DETAILS		
	HIGH	MEDIUM	LOW	INFO	OPT	ROWS	SIZE	NUMS
Dataset_1	13,149	35,155	10,175	19,659	38,314	12,412,520	433.3MB	38,600
Dataset_2	12	37	54	-	-	-	6.3MB	579
Dataset_3	-	-	-	-	-	7,954,979	284.3MB	13,413
Dataset_4	-	-	-	-	-	-	1.295GB	183,710
Dataset_5	-	-	-	-	-	7,565	273.9KB	29

4.1.4 Metrics

We define the discovery of vulnerable contracts as a problem. By comparing the methods' detection results with the previous vulnerability labels, we can measure whether the problem occurs, which can be regarded as a binary classification. In this way, all problems found by methods are marked as true positive (TP), false positive (FP), true negative (TN) and false negative (FN). TP and TN indicate the results which correctly predict a contract with and without a vulnerability. In contrast, FP and FN describe false detection. Furthermore, the accuracy (ACC), precision (P), recall (R), and F-Measure (F1) are calculated as follows to evaluate each method, where $\#TP$, $\#TN$, $\#FP$, and $\#FN$ refer to the number of contracts marked accordingly.

$$ACC = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN}$$

$$P = \frac{\#TP}{\#TP + \#FP}, R = \frac{\#TP}{\#TP + \#FN}, F1 = \frac{2 \times P \times R}{P + R}$$

4.1.5 Parameter settings

Without special mention in texts, we report the performance of all models with the following empirical settings: $n_{\text{cycle}} = 2$, $n_{\text{block}} = 32$, $n_{\text{seq}} = 10$, $T = 512$, $\lambda = 0.6$, $q_b = 0.8$, $q_m = 0.2$,

9. The number of each vulnerability is detailed in https://github.com/ContractAudit/VulHunter/tree/main/Dataset_vul_num.

$epoch = 50$, $n_{\text{neurons}} = 512$, and the instances with the longer lengths. Also, the Bi^2-LSTM is selected by default, and the baseline models and hyperparameters are evaluated in § 4.6 and § 4.7. For ML-based methods on Datasets_1~2, in order to explore the effect of different proportions between benign and malicious contracts, we randomly select 80% of benign/malicious contracts as the training dataset and the other 20% as the testing dataset based on the five random seeds (*i.e.,* 42, 1234, 2345, 3456, and 4567) respectively, and report the averaged results with standard deviations. Furthermore, the Bi^2-LSTM model trained on Datasets_1~2 was employed to identify the contracts in Datasets_3~5.

4.2 Precision of VulHunter (RQ1)

To answer the first research question, we compared the ability of VulHunter with SOTA methods based on traditional analysis and ML to detect contract source code in Dataset_1 and bytecode in Dataset_2. Specifically, (i) we executed the methods on these contracts.[10] (ii) We extracted all vulnerabilities detected by methods into JSON files. (iii) We transformed each vulnerability category detected by the method into a pre-uniform name, as described in Table 4. For instance, SmartCheck detects a vulnerability called *SOLIDITY_TX_ORIGIN* that we link to the *tx-origin* (TO) category. (iv) We used the true labels of contracts to calculate the metrics based on the detection results.

In order to balance the training dataset and evaluate the effect of different ratios between the benign and malicious contracts on VulHunter performance, we set two proportions of 2:1 and 5:1 for considering the minimum requirements of the sample numbers. The results are presented in Tables 6~7, which illustrate the performance for each method.[11] Note that methods such as SmartCheck have no standard deviations given their deterministic detection. Both tables contain the three parts: (i) metrics such as ACC and F1 of some vulnerabilities (*e.g., reentrancy-eth*); (ii) the total AVG and NAVG metrics of each vulnerability severity, where the NAVG only covers the vulnerability categories that methods can identify; (iii) the AVG and NAVG metrics of each method.

First of all, in the 2:1 experiment, we can summarize from part (i): VulHunter overperforms other methods for most vulnerabilities, such as *controlled-delegatecall* and *timestamp*. In contrast, the performance of traditional detection methods such as SmartCheck and Securify is restrained by their unrenewed rules. On the one hand, this reflects

10. The source code and execution result of contracts are available in https://github.com/ContractAudit/VulHunter/tree/main/Dataset1.

11. The specific other results are detailed in https://github.com/ContractAudit/VulHunter/tree/main/Dataset1/Detection_result.

TABLE 6
Comparative results detected by each method on Dataset_1 (benign:vulnerable = 2:1). The forms of bold and underlines highlight the best and second results for each project, respectively. Note that "SE." is the abbreviation of "Severity", and "-" means the project is not supported. Also, "Total NAVG (Net average)" refers to the average results of vulnerabilities supported by each method, while "Total AVG" represents that of all vulnerabilities.

SE.	Project	Metrics	VulHunter	SmartCheck	Slither	Securify	Oyente	Mythril	DefectChecker	SMARTIAN	TMP	ContractWard
High	RE	ACC P	95.29±0.78 95.06±0.82	- -	98.04 94.44	72.55 66.67	66.27 49.06	48.24 3.92	73.73 82.14	66.27 0.00	93.73±0.81	96.00±0.60 81.18±1.58 93.02±1.86
		R F1	90.59±1.66 92.77±1.24	- -	100.00 97.14	35.29 46.15	30.59 37.68	2.35 2.94	27.06 40.71	0.00 0.00	84.71±2.02	90.00±1.38 47.06±4.05 62.50±3.93
	CDC	ACC P	**94.87**±2.05 92.31±3.08	- -	94.87 100.00	- -	- -	53.85 27.27	- -	64.10 0.00	82.05±2.51	68.75±3.45 66.67±0.00 0.00±0.00
		R F1	**92.31**±3.08 **92.31**±3.08	- -	84.62 91.67	- -	- -	23.08 25.00	- -	0.00 0.00	84.62±3.08	75.86±3.21 0.00±0.00 0.00±0.00
	Total	ACC P	**89.61**±1.98 **85.35**±2.90	- -	88.33 81.76	83.61 76.44	64.56 46.21	55.29 24.68	73.73 82.14	67.15 25.00	82.25±4.03	80.92±7.69 71.25±0.39 65.46±2.71
	NAVG	R F1	**83.43**±3.14 **84.38**±3.06	- -	66.46 73.32	67.65 71.78	33.91 39.11	16.17 19.54	27.06 40.71	4.49 7.61	64.60±8.73	71.84±8.63 14.51±1.03 23.75±1.04
Medium	LE	ACC P	88.03±2.02 80.54±3.14	73.58 67.12	97.17 99.87	62.90 46.93	- -	- -	64.35 36.77	65.84 0.00	86.26±7.00	95.95±3.88 79.43±0.12 98.80±0.2
		R F1	84.53±2.61 82.49±2.89	40.64 50.62	91.64 95.58	86.34 60.80	- -	- -	9.66 15.30	0.00 0.00	61.37±20.78	74.86±20.28 38.75±0.28 55.67±0.32
	UCL	ACC P	92.31±1.08 87.50±1.58	84.62 100.00	97.44 100.00	88.89 96.43	- -	60.68 33.33	93.16 100.00	80.34 100.00	82.91±2.23	70.21±3.38 66.67±0.00 0.00±0.00
		R F1	89.74±1.62 88.61±1.60	53.85 70.00	92.31 96.00	69.23 80.60	- -	17.95 23.33	79.49 88.57	41.03 58.18	84.62±1.62	76.74±2.66 0.00±0.00 0.00±0.00
	Total	ACC P	**90.78**±1.70 88.99±2.08	82.13 91.41	90.45 93.70	75.51 68.53	76.74 82.25	46.62 22.59	84.16 86.63	67.34 74.97	82.00±3.23	76.91±4.69 71.09±1.87 63.22±1.17
	NAVG	R F1	**86.31**±2.34 **87.63**±2.35	51.41 65.81	72.00 81.43	72.60 70.51	90.25 86.06	10.24 14.09	56.30 68.24	29.82 42.67	69.73±6.86	73.14±6.84 21.44±2.68 32.02±2.64
	Total	ACC P	**90.78**±1.70 **88.99**±2.08	36.50 40.63	80.40 83.28	25.17 22.84	8.53 9.14	20.72 10.04	46.76 48.13	37.41 41.65	82.00±3.23	76.91±4.69 71.09±1.87 63.22±1.17
	AVG	R F1	**86.31**±2.34 **87.63**±2.35	22.85 29.25	64.00 72.38	24.20 23.50	10.03 9.56	4.55 6.26	31.28 37.91	16.57 23.70	69.73±6.86	73.14±6.84 21.44±2.68 32.02±2.64
Low	TS	ACC P	**89.61**±1.72 **89.26**±1.94	66.91 100.00	77.54 59.91	- -	59.66 35.05	- -	- -	65.94 0.00	84.06±3.22	80.00±4.66 72.22±0.42 96.00±2.53
		R F1	**78.26**±3.71 **83.40**±2.95	0.72 1.44	98.55 74.52	- -	24.64 28.94	- -	- -	0.00 0.00	69.57±6.17	74.42±5.55 17.39±0.85 29.45±1.31
	BP	ACC P	**88.14**±1.99 86.34±2.58	- -	- -	- -	- -	51.56 15.09	88.00 99.65	65.83 7.69	81.40±3.90	79.39±5.82 69.48±0.36 91.11±2.02
		R F1	**76.48**±3.96 **81.11**±3.36	- -	- -	- -	- -	9.79 11.88	64.24 78.12	0.23 0.44	59.68±8.47	68.14±7.66 9.34±0.94 16.94±1.58
	Total	ACC P	**91.00**±1.51 88.87±1.98	79.06 99.92	86.07 79.85	- -	59.66 35.05	51.56 15.09	74.98 93.01	65.90 2.56	82.32±4.15	76.80±6.16 70.78±0.32 96.05±1.24
	NAVG	R F1	**83.41**±2.81 **86.05**±2.45	37.25 54.27	91.27 85.18	- -	24.64 28.94	9.79 11.88	25.32 39.81	0.08 0.15	67.79±7.44	72.01±6.98 12.86±0.81 22.68±1.30
Info	LLC	ACC P	**90.61**±1.60 88.31±2.13	68.98 100.00	84.49 99.82	- -	- -	- -	- -	- -	84.25±6.65	84.44±9.23 76.98±0.21 98.79±0.12
		R F1	**82.83**±2.92 **85.48**±2.55	6.97 13.03	53.58 69.73	- -	- -	- -	- -	- -	64.68±18.18	73.25±16.55 31.33±0.60 47.57±0.70
	Total	ACC P	90.62±1.60 86.86±2.22	87.94 100.00	92.25 99.91	- -	- -	- -	- -	- -	81.78±3.95	78.35±5.01 72.29±0.23 49.20±0.07
	NAVG	R F1	85.25±2.49 86.05±2.46	64.24 78.23	76.79 86.84	- -	- -	- -	- -	- -	69.56±7.78	73.70±7.33 18.05±0.67 26.41±0.69
	Total	ACC P	**90.62**±1.60 86.86±2.22	87.94 100.00	46.12 49.96	- -	- -	- -	- -	- -	81.78±3.95	78.35±5.01 72.29±0.23 49.20±0.07
	AVG	R F1	**85.25**±2.49 **86.05**±2.46	64.24 78.23	38.40 43.42	- -	- -	- -	- -	- -	69.56±7.78	73.70±7.33 18.05±0.67 26.41±0.69
Opt	ST	ACC P	85.42±2.57 82.69±3.58	100.00 100.00	- -	- -	- -	- -	- -	- -	77.72±8.92	68.52±24.28 66.67±0.00 0.00±0.00
		R F1	71.27±5.03 82.69±4.44	100.00 100.00	- -	- -	- -	- -	- -	- -	61.33±19.72	64.72±21.25 0.00±0.00 0.00±0.00
	Total	ACC P	88.38±2.50 89.37±2.10	91.67 94.85	90.07 74.43	- -	- -	- -	- -	- -	80.85±4.63	83.53±6.90 65.18±0.67 62.55±0.90
	NAVG	R F1	77.82±5.15 83.20±4.59	92.16 93.48	71.34 72.85	- -	- -	- -	- -	- -	62.59±10.19	71.56±10.11 10.83±1.01 18.47±1.24
	Total	ACC P	**88.38**±2.50 **89.37**±2.10	45.83 47.43	60.05 49.62	- -	- -	- -	- -	- -	80.85±4.63	83.53±6.90 65.18±0.67 62.55±0.90
	AVG	R F1	**77.82**±5.15 **83.20**±4.59	46.08 46.74	47.56 48.57	- -	- -	- -	- -	- -	62.59±10.19	71.56±10.11 10.83±1.01 18.47±1.24
Overall	Total	ACC P	**90.04**±1.89 **87.92**±2.28	85.17 96.43	89.57 86.24	78.75 71.70	66.38 52.43	50.49 22.44	79.94 **88.26**	66.92 40.21	81.84±3.92	79.35±6.07 70.06±0.86 66.12±1.34
	NAVG	R F1	**83.41**±3.17 **85.60**±2.99	60.77 74.56	72.56 78.81	70.62 71.15	45.68 48.82	12.41 15.98	42.72 57.57	13.94 20.70	66.82±8.16	72.55±8.00 16.10±1.44 25.90±1.55
	Total	ACC P	**90.04**±1.89 **87.92**±2.28	39.75 45.00	65.68 63.25	13.12 11.95	8.85 6.99	13.46 5.98	23.98 26.48	26.77 16.08	81.84±3.92	79.35±6.07 70.06±0.86 66.12±1.34
	AVG	R F1	**83.41**±3.17 **85.60**±2.99	28.36 34.79	53.21 57.79	11.77 11.86	6.09 6.51	3.31 4.26	12.82 17.27	5.58 8.28	66.82±8.16	72.55±8.00 16.10±1.44 25.90±1.55
	#Failed NUMS		**0(0%)**	0(0%)	17(0.04%)	229(0.59%)	6,275(16.26%)	8,490(21.99%)	26(0.07%)	395(1.02%)	0(0%)	0(0%)

that the contract expression difference caused by the vulnerability evolution and the compiler upgrades can bypass their fixed detection rules. This reminds their experts to constantly develop and update the detection rules to stay current, which is quite cumbersome. On the other hand, it indicates that VulHunter can automatically capture the effective vulnerability features based on datasets through the Bi^2-LSTM model, given its data representations/fitting and temporal contextual correlation ability for contract instances. Nonetheless, there are some vulnerabilities such as *reentrancy-eth*, VulHunter is slightly insufficient to the methods such as Slither. It is because these vulnerabilities can be deterministically described by pre-defined rules, which can enable sophisticated detection by using pattern matching. This inspires ML-based methods including VulHunter to incorporate some deterministic knowledge, similar to AME [49] and CGE [5] Particularly, due to the variability and diversity of ML, we can enhance the performance of VulHunter by adjusting some variable parameters and baseline models, which is discussed in § 4.6 and § 4.7. For example, increasing the number of extracted instances can make the model detect more paths, facilitating vulnerability discovery. More importantly, VulHunter does not rely on manual predefined rules, making it easier to detect new vulnerabilities without the involvement of experts. This is one of the reasons why we used ML models rather than traditional detection logic. Notably, it identifies (almost) all vulnerabilities better than other ML-based methods such as TMP and ContractWard. This stems from its ability to focus on the runtime execution sequences (similar to symbolic execution-based methods such as DefectChecker) and accurately capture the subtle features of benign and malicious samples during the contract execution. In contrast, TMP and ContractWard are insensitive to vulnerability features by globally observing the generalized contract bytecode, holding the inferior recall rate and F1 score. In addition, some vulnerabilities or defects such as *uninitialized-state* and *unused-state* are challenging to identify at the bytecode level, which is discussed in § 5.2.1.

By observing the severity from *Opt* to *High* (*i.e.*, part (ii)), the detection performance of VulHunter is almost the best compared to others. While some traditional methods, such as Slither and DefectChecker, can only work well with their supported vulnerabilities. For example, Slither just identifies two problems with *Info* severity and achieves ACC=92.25% and P=99.91%, indicating that these defects are adequately described by their rules. Nevertheless, it is difficult to develop well-established rules for complex vulnerabilities such as *tod* as they cannot execute contracts and account for all situations. Also, massive detection rules manually developed by experts for each vulnerability are frequent and time-consuming,

TABLE 7
Comparative results detected by each method on Dataset_1 (benign:vulnerable = 5:1).

SE.	Project	Metrics	VulHunter	SmartCheck	Slither	Securify	Oyente	Mythril	DefectChecker	SMARTIAN	TMP	ContractWard
High	RE	ACC P	96.46±0.61 90.36±1.75	- -	99.02 94.44	81.18 37.78	73.53 16.22	61.37 0.88	84.31 58.06	81.76 0.00	94.51±0.67 89.04±1.76	86.67±0.28 86.96±1.92
		R F1	88.24±2.02 89.29±1.88	- -	100.00 97.14	20.00 26.15	14.12 15.09	1.18 1.01	21.18 31.03	0.00 0.00	76.47±2.68 82.28±2.29	23.53±1.37 37.04±1.82
	CDC	ACC P	**96.15**±0.51 **100.00**±0.00	- -	96.15 100.00	- -	- -	71.79 15.38	- -	83.33 0.00	91.03±0.81 100.00±0.00	83.33±0.00 0.00±0.00
		R F1	**76.92**±3.08 **86.96**±2.06	- -	76.92 86.96	- -	- -	15.38 15.38	- -	0.00 0.00	46.15±4.87 63.16±4.57	0.00±0.00 0.00±0.00
	Total	ACC P	92.75±1.48 **83.16**±3.94	- -	93.94 80.88	87.08 55.68	70.67 20.90	65.89 9.51	84.31 58.06	83.33 21.88	89.32±1.32 82.64±5.13	84.58±0.10 53.89±0.40
	NAVG	R F1	**73.38**±5.31 **77.96**±4.84	- -	65.57 72.42	60.00 57.76	24.08 22.38	11.50 10.41	21.18 31.03	4.49 7.45	48.88±6.66 61.43±6.57	10.77±0.52 17.96±0.58
	Total	ACC P	**92.75**±1.48 **83.16**±3.94	- -	80.52 69.32	24.88 15.91	20.19 5.97	28.24 4.08	12.04 8.29	47.62 12.50	89.32±1.32 82.64±5.13	84.58±0.10 53.89±0.40
	AVG	R F1	**73.38**±5.31 **77.96**±4.84	- -	56.20 62.08	17.14 16.50	6.88 6.39	4.93 4.46	3.03 4.43	2.56 4.26	48.88±6.66 61.43±6.57	10.77±0.52 17.96±0.58
Medium	LE	ACC P	93.06±4.26 87.09±11.26	83.08 49.08	98.59 99.74	55.50 25.09	- -	- -	77.03 16.63	82.08 0.00	93.15±3.73 90.85±7.79	89.14±0.03 98.68±0.16
		R F1	68.56±23.99 76.72±22.13	40.75 44.53	91.76 95.58	84.10 38.65	- -	- -	9.42 12.03	0.00 0.00	65.49±21.55 76.11±20.39	35.34±0.17 52.04±0.20
	UCL	ACC P	94.87±0.88 96.55±1.61	91.45 91.30	97.86 100.00	90.17 78.57	- -	68.38 11.11	92.31 80.00	88.46 83.33	90.17±0.97 86.36±1.44	83.33±0.00 0.00±0.00
		R F1	71.79±4.41 82.35±3.42	53.85 67.74	87.18 93.15	56.41 65.67	- -	12.82 11.90	71.79 75.68	38.46 52.63	48.72±5.85 62.30±5.16	0.00±0.00 0.00±0.00
	Total	ACC P	92.87±1.84 86.25±4.38	89.84 83.14	92.99 93.59	75.22 49.01	76.74 82.25	53.29 11.82	89.93 78.37	75.23 70.57	86.19±2.42 77.17±5.58	81.49±1.56 43.39±0.32
	NAVG	R F1	79.15±6.49 82.55±5.99	51.11 63.30	71.75 81.23	67.92 56.93	90.25 86.06	6.54 8.42	53.35 63.49	26.79 38.84	61.45±7.47 68.42±7.70	17.43±2.21 24.87±2.18
	Total	ACC P	**92.87**±1.84 **86.25**±4.38	39.93 36.95	82.66 83.19	25.07 16.34	8.53 9.14	23.68 5.25	49.96 43.54	41.80 39.20	86.19±2.42 77.17±5.58	81.49±1.56 43.39±0.32
	AVG	R F1	**79.15**±6.49 **82.55**±5.99	22.71 28.13	63.78 72.20	22.64 18.98	10.03 9.56	2.90 3.74	29.64 35.27	14.88 21.58	61.45±7.47 68.42±7.70	17.43±2.21 24.87±2.18
Low	TS	ACC P	**95.15**±0.83 83.56±2.61	83.56 100.00	76.78 41.72	- -	69.29 15.88	- -	- -	81.98 0.00	93.11±0.89 84.62±2.50	83.31±0.00 0.00±0.00
		R F1	88.41±2.05 **85.92**±2.34	1.45 2.86	98.55 58.62	- -	19.57 17.53	- -	- -	0.00 0.00	71.74±3.50 77.65±3.10	0.00±0.00 0.00±0.00
	BP	ACC P	89.19±4.47 67.92±20.46	- -	- -	- -	- -	62.78 7.54	93.62 98.22	82.34 0.00	88.61±1.50 80.09±5.13	83.33±0.00 0.00±0.00
		R F1	**66.36**±18.91 67.13±19.19	- -	- -	- -	- -	10.93 8.92	62.87 76.67	0.00 0.00	42.14±7.43 55.22±7.49	0.00±0.00 0.00±0.00
	Total	ACC P	**93.36**±1.79 82.43±6.78	89.87 99.60	87.01 70.86	- -	69.29 15.88	62.78 7.54	85.64 89.87	82.28 0.00	89.01±2.14 80.02±10.47	82.49±0.01 47.95±0.42
	NAVG	R F1	79.27±6.84 **80.82**±6.72	39.38 56.45	91.01 79.68	- -	19.57 17.53	10.93 8.92	25.05 39.18	0.00 0.00	52.47±10.29 63.38±11.11	2.52±0.01 4.80±0.02
Info	LLC	ACC P	**92.41**±1.29 89.57±1.95	74.93 100.00	87.46 99.82	- -	- -	- -	- -	- -	87.27±4.00 81.05±6.50	80.24±0.07 99.12±0.08
		R F1	**81.39**±**3.17** **85.28**±**2.62**	6.97 13.03	53.58 69.73	- -	- -	- -	- -	- -	68.86±12.63 74.46±10.71	26.91±0.25 42.33±0.31
	Total	ACC P	91.37±1.91 84.60±4.53	90.22 93.75	93.73 99.91	- -	- -	- -	- -	- -	89.92±1.71 87.45±2.77	82.39±0.17 49.46±0.04
	NAVG	R F1	72.60±7.46 78.14±6.43	59.24 72.60	76.79 86.84	- -	- -	- -	- -	- -	64.83±5.84 74.46±5.01	17.34±0.58 25.68±0.59
	Total	ACC P	**91.37**±1.91 84.60±4.53	90.22 93.75	46.87 49.96	- -	- -	- -	- -	- -	89.92±1.71 87.45±2.77	82.39±0.17 49.46±0.04
	AVG	R F1	**72.60**±7.46 **78.14**±6.43	59.24 72.60	38.40 43.42	- -	- -	- -	- -	- -	64.83±5.84 74.46±5.01	17.34±0.58 25.68±0.59
Opt	ST	ACC P	88.25±4.29 64.21±16.22	100.00 100.00	- -	- -	- -	- -	- -	- -	88.10±3.59 69.70±28.85	83.30±0.00 0.00±0.00
		R F1	67.40±15.73 65.77±15.31	100.00 100.00	- -	- -	- -	- -	- -	- -	50.83±18.50 58.79±21.72	0.00±0.00 0.00±0.00
	Total	ACC P	**92.19**±2.08 86.22±4.71	91.83 97.32	94.99 74.58	- -	- -	- -	- -	- -	88.81±1.55 78.91±7.02	80.51±2.45 32.13±0.90
	NAVG	R F1	72.76±8.11 78.92±7.41	92.45 94.82	71.53 73.02	- -	- -	- -	- -	- -	65.15±6.55 71.37±7.25	12.81±2.99 18.32±3.09
	Total	ACC P	**92.19**±2.08 **86.22**±4.71	45.92 48.66	63.32 49.72	- -	- -	- -	- -	- -	88.81±1.55 78.91±7.02	80.51±2.45 32.13±0.90
	AVG	R F1	**72.76**±8.11 **78.92**±7.41	46.23 47.41	47.69 48.68	- -	- -	- -	- -	- -	65.15±6.55 71.37±7.25	12.81±2.99 18.32±3.09
Overall	Total	ACC P	92.57±1.81 84.79±4.68	90.38 92.74	93.14 85.17	79.97 51.68	71.84 34.98	59.20 10.42	87.87 79.95	79.70 36.69	88.32±1.86 80.55±6.04	82.27±1.00 45.01±0.43
	NAVG	R F1	**75.67**±6.72 **79.97**±6.16	59.78 72.70	72.23 78.17	64.75 57.48	39.49 37.10	8.95 9.63	40.34 53.63	12.66 18.82	58.51±7.26 67.78±7.44	12.96±1.46 20.12±1.49
	Total	ACC P	**92.57**±1.81 **84.79**±4.68	42.18 43.28	68.30 62.46	13.33 8.61	9.58 4.66	15.79 2.78	26.36 23.98	31.88 14.68	88.32±1.86 80.55±6.04	82.27±1.00 45.01±0.43
	AVG	R F1	**75.67**±6.72 **79.97**±6.16	27.90 33.93	52.97 57.33	10.79 9.58	5.27 4.95	2.39 2.57	12.10 16.09	5.06 7.53	58.51±7.26 67.78±7.44	12.96±1.46 20.12±1.49
#Failed NUMS			**0(0%)**	0(0%)	17(0.04%)	229(0.59%)	6,275(16.26%)	8,490(21.99%)	26(0.07%)	395(1.02%)	0(0%)	0(0%)

restricting the vulnerabilities they can detect. Therefore, they have an inferior total AVG for each severity. Besides, detection rules based on the bytecode are more difficult to develop than the source code, given their different readability. This is why source code detection methods such as Slither can identify more vulnerability types than those based on the bytecode, *e.g.,* Securify. Conversely, VulHunter overcomes the bottleneck of bytecode-based rule-making with its keen feature observation and representation on massive datasets, thus achieving or exceeding the detection performance of source code-based methods without expert involvement.

From part (iii), it is concluded that VulHunter can detect contracts more accurately, discover most vulnerabilities, and perform with an acceptable standard deviation below TMP, due to its effective information extraction and tailored multi-attention mechanism. Notably, VulHunter can choose suitable random seeds to achieve better results, which is impossible for traditional methods. Also, some contracts cannot be analyzed by other arts. Even Oyente and Mythril hold failure rates of 16.26% and 21.99%, respectively. This may be a reason for their poor detection. Instead, VulHunter can analyze more contracts given its robust implementation and refined model.

From Table 7, we can draw similar conclusions as above. VulHunter can still detect more vulnerabilities and overperform other methods. Since the proportion of the training set grows 5:1, *i.e.,* the benign samples are expanded, the model is inclined to learn benign features, thereby improving the identification accuracy and reducing the standard deviation, *i.e.,* the overall ACC=92.57% (2.33%↑) and STD_{ACC}=1.81% (0.08%↓). Nonetheless, the diluted malicious features enable the model convergence biased to benign samples, reducing its recall rate and increasing the standard deviation. Although other methods may not require pre-training, this phenomenon also applies to them due to the dataset variation. Even the changes in the data distribution has seriously affected the performance of ContractWard, such as the failure for the *timestamp* vulnerability identification (P=0 and R=0). This reflects that their robustness needs to be improved. Thus, the auditors can sample in appropriate proportions based on their identification requirements and the contract distribution in collected dataset, so as to achieve their specific effects. For example, reporting suspected vulnerabilities as much as possible can set a small ratio between benign and malicious contracts to perform a superior recall rate. Instead, they can set a large proportion to detect more contracts correctly, thereby relieving the pressure on manual review.

Second, we utilized 7 methods supporting bytecode analysis to detect the contracts in Dataset_2. As described in Table 8, the results agree with the above conclusion that VulHunter can detect most vulnerabilities accurately. Also, although the individual detection result changes in small datasets can lead to significant differences, VulHunter still keeps acceptable standard deviations, demonstrating its stability. Notably, for vulnerabilities such as *reentrancy-eth*, symbolic execution-based methods (*i.e.,* DefectChecker, Oyente, and Mythril) achieve superior precision by executing symbolic inference, and DefectChecker makes the best performance. Nevertheless, its inferior recall rate may be affected by incomplete pre-defined rules, reflecting that it is difficult to cover all cases given their massive number and unreadable bytecode. In contrast, VulHunter performs the accurate model inference to automatically capture the semantic features of benign and vulnerable contracts, thus considering both precision and recall, and holding a superior F1 score. Moreover, Securify and SMARTIAN failed to analyze most contracts due to procedural errors. They can improve the performance by optimizing their implementation code. The imperfect features of ContractWard lead to its limited analysis and zero fluctuations, suggesting its model needs to be refined to focus on subtle runtime features. In a word, VulHunter can perform a superior detection based on bytecode due to its actual execution path inputs, well-

TABLE 8
Comparative results of ACC, P, R, and F1 detected by each tool on Dataset_2 (benign:vulnerable = 2:1).

SE.	Project	VulHunter				Securify				Oyente				Mythril				DefectChecker				SMARTIAN				ContractWard			
		ACC	P	R	F1	ACC	P	R	F1	ACC	P	R	F1	ACC	P	R	F1	ACC	P	R	F1	ACC	P	R	F1	ACC	P	R	F1
High	RE	**100.00** ±0.00	**100.00** ±0.00	**100.00** ±0.00	**100.00** ±0.00	41.67	23.53	33.33	27.59	77.78	100.00	33.33	50.00	66.67	0.00	0.00	0.00	88.89	100.00	66.67	80.00	0.00	0.00	0.00	0.00	66.67 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
Medium	LE	94.12 ±3.72	83.33 ±9.11	**100.00** ±0.00	**90.91** ±5.28	61.11	44.44	66.67	53.33	-	-	-	-	-	-	-	-	94.44	100.00	83.33	90.91	0.00	0.00	0.00	0.00	33.33 ±0.00	33.33 ±0.00	100.00 ±0.00	50.00 ±0.00
	TO	**93.33** ±4.22	83.33 ±9.11	**100.00** ±0.00	**90.91** ±5.28	-	-	-	-	-	-	-	-	80.00	100.00	40.00	57.14	93.33	100.00	80.00	88.89	0.00	0.00	0.00	0.00	33.33 ±0.00	33.33 ±0.00	100.00 ±0.00	50.00 ±0.00
	IE	**100.00** ±0.00	**100.00** ±0.00	**100.00** ±0.00	**100.00** ±0.00	-	-	-	-	-	-	-	-	-	-	-	-	73.33	100.00	20.00	33.33	-	-	-	-	33.33 ±0.00	33.33 ±0.00	100.00 ±0.00	50.00 ±0.00
	UCL	**95.45**	95.24	**90.91**	**93.02**	50.00	33.33	50.00	40.00	-	-	-	-	74.24	100.00	22.73	37.04	84.85	92.86	59.09	72.22	0.00	0.00	0.00	0.00	66.67	0.00	0.00	0.00
	UCS	±1.92	±3.01	±2.87	±2.94					-	-	-	-													±0.00	±0.00	±0.00	±0.00
	Total	**95.73**	90.48	**97.73**	**93.96**	55.56	38.89	58.33	46.67	-	-	-	-	77.12	100.00	31.36	47.75	86.49	98.21	60.61	74.96	0.00	0.00	0.00	0.00	41.67	25.00	75.00	37.50
	NAVG	±2.46	±5.31	±0.72	±3.38					-	-	-	-													±0.00	±0.00	±0.00	±0.00
Low	BP,TS	**95.97** ±1.61	93.02 ±3.15	**95.24** ±1.51	**94.12** ±2.30	-	-	-	-	61.90	31.25	11.90	17.24	64.29	40.00	14.29	21.05	94.44	100.00	83.33	90.91	0.79	0.00	0.00	0.00	66.67 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	CTL	**89.74** ±5.13	90.91 ±6.36	**76.92** ±10.88	**83.33** ±8.91	-	-	-	-	-	-	-	-	-	-	-	-	71.79	100.00	15.38	26.67	-	-	-	-	66.67 ±0.00	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00
	CLL	88.89 ±7.03	83.33 ±10.54	83.33 ±10.54	83.33 ±10.54	-	-	-	-	-	-	-	-	-	-	-	-	94.44	100.00	83.33	90.91	0.00	0.00	0.00	0.00	33.33 ±0.00	33.33 ±0.00	100.00 ±0.00	50.00 ±0.00
	Total	**91.53**	89.09	**85.16**	**87.08**	-	-	-	-	61.90	31.25	11.90	17.24	64.29	40.00	14.29	21.05	86.89	100.00	60.68	75.53	0.40	0.00	0.00	0.00	55.56	11.11	33.33	16.67
	NAVG	±4.59	±6.68	±7.64	±7.25	-	-	-	-																	±0.00	±0.00	±0.00	±0.00
Overall	Total	**94.69**	91.15	**93.30**	**92.21**	50.93	33.77	50.00	40.31	69.84	65.63	22.62	33.64	71.30	60.00	19.25	29.15	86.94	99.11	61.39	75.82	0.13	0.00	0.00	0.00	50.00	16.67	50.00	25.00
	NAVG	±2.95	±5.16	±3.23	±4.41																					±0.00	±0.00	±0.00	±0.00
	#Failed		**0(0%)**			146(25.22%)				0(0%)				0(0%)				0(0%)				578(99.83%)				0(0%)			

TABLE 9
Comparison of vulnerability number detected on Dataset_3.

SE.	Project	VulHunter	SmartCheck	Slither	Securify	Oyente	Mythril	DefectChecker	SMARTIAN	TMP	ContractWard
High	RE	1,418	-	295	375	12	11	362	0	1,294	4,364
	SU	3,374	-	77	-	-	25	-	16	3,156	0
	CDC	3,797	-	109	-	-	19	-	0	6,453	0
	Total sum	21,311	0	2,740	3238	246	55	362	16	21,020	4,392
Medium	LE	3,022	2,037	1,484	5,827	-	-	4,296	0	374	45
	TO	1,884	383	117	-	-	19	279	19	4,551	0
	UCS	2,207	-	127	1321	-	1	1,257	220	705	297
	Total sum	28,459	2,690	4,530	7,394	5,058	69	7,170	712	35,665	2,859
Low	TS	968	26	3,147	-	73	-	-	1	2,784	1,368
	BP	1,522	-	-	-	-	130	680	1	2,043	70
	CLL	3,020	151	1,060	-	-	-	167	0	3,311	115
	Total sum	9,697	3,572	4,207	0	73	130	875	2	11,955	1,570
Info	LLC	3,039	1,696	4,454	-	-	-	-	-	2,956	78
	E20ID	5,787	19	17	-	-	-	-	-	1,895	0
	HC	3,185	5,403	-	-	-	-	-	-	3,519	1,627
	Total sum	16,706	7,277	4,471	0	0	0	0	0	8,462	1,705
Opt	COL	2,440	-	130	-	-	-	-	-	671	1
	ST	1,662	83	-	-	-	-	-	-	3,486	0
	BE	1,590	-	1,432	-	-	-	-	-	435	1,659
	Total sum	22,203	11,759	15,085	0	0	0	0	0	16,970	4,133
Overall	Total sum	98,376	25,298	31,033	10,632	5,377	254	8,407	730	94,072	14,659
	#Failed	0(0%)	0(0%)	191(1.42%)	235(1.75%)	8,742(65.18%)	227(1.69%)	749(5.58%)	1,327(9.89%)	0(0%)	0(0%)
	#Secure	28	570	416	5,946	1,361	12,945	6,864	11,580	1	5,955

designed model, and meticulous prototype implementation.

Answer to RQ1. What is the effectiveness of VulHunter in detecting contract vulnerabilities? Compared with the SOTA methods on multiple datasets, VulHunter has a superior contract detection performance and acceptable standard deviations (ACC=90.04%, P=87.92%, R=83.41% and $F1$=85.60%). Also, it can detect more contracts normally (Failed=0%), which illustrates its robust implementation. More importantly, unlike traditional methods, it can analyze source code and bytecode accurately without manual pre-defined rules, thus easily expanding to identify new vulnerabilities and consider the complex representations caused by vulnerability evolution and compiler updates. In turn, its identified defective source code statements and key opcode subsequences can help them maintain detection rules to consider more cases and improve the performance.

4.3 Vulnerabilities in Production Smart Contracts (RQ2)

To answer the second research question, we ran the 10 methods on contract source code in Dataset_3 and bytecode in Dataset_4. The detailed results are given in Table 9~10, which aim to show the frequency of each vulnerability on Ethereum. Specifically, Table 9 shows the detection situation of various severity for each method.[12] Among them, VulHunter and TMP detected the most contract vulnerabilities. It can be attributed to the most vulnerability categories they detected and the comprehensive vulnerability features they learned, which require laborious development by many contract experts for traditional methods. Specifically, VulHunter discovered 443 contracts without vulnerabilities exceeding *Info* severity, and more than half of the contracts have multiple problems. Note that the higher number may be caused by a few vulnerabilities, *e.g.*, the *integer-overflow* accounts for 35.3% of *Medium* severity. Although, as mentioned in [51], [52], many discovered defects with advanced utilization difficulties or even not exploitable in practice (*i.e.,* the proportions of probably and possibly in each severity are *High* (67.2%), *Medium* (65.2%), *Low* (100.0%), respectively), identifying these problems can draw the attention of contract developers to possible threats, thereby reducing the risk of contracts being attacked. Moreover, numerous contracts analysis failed by Oyente and SMARTIAN, which may be a significant factor restraining their vulnerability discovery.

As conferred in § 4.2 and § 4.6~4.7, auditors can adjust the contract proportion of the training dataset, baseline models (*e.g.,* Decision Tree and Random Forest), and hyperparameters (*e.g.*, q_m and q_b) to determine whether VulHunter focuses on vulnerability discovery or benign contract identification. In this section, we employed the settings of ratio=2:1, Bi^2-LSTM model, q_b=0.8, and q_m=0.2 to detect contract security threats as much as possible. Nevertheless, this inevitably produces some false positives. To this end, as detailed in 3.7, on the one hand, VulHunter outputs defective source code statements and key opcode fragments to facilitate the manual secondary verification and estimate false reports quickly. On the other hand, it can employ the optional constraint-solving module to automatically validate the feasibility of vulnerable instances and reduce the workload of manual verification, which is explored in § 4.9. Note that the latter cannot be provided by other methods based on static analysis and ML. Besides, some contracts are discussed in § 4.5 to illustrate the validity of partial results. Overall, given the superior detection capabilities of VulHunter, the security of contracts on Ethereum needs to be taken seriously, and owners should develop contracts based on the specifications [53].

Due to the expansive time overhead in other methods such as Mythril and Oyente, we randomly sampled and detected 10,000 bytecodes from the Dataset_4, and all of the 183,710 bytecodes were identified only by VulHunter.[13] Table 10 describes similar phenomena as mentioned above. Since the number of failed contracts of methods such as Securify and SMARTIAN increases, their results are incom-

12. The specific detection results can be found in https://github.com/ContractAudit/VulHunter/tree/main/Dataset3/Detection_result.

13. All results were detailed in https://github.com/ContractAudit/VulHunter/tree/main/Dataset4/Detection_result.

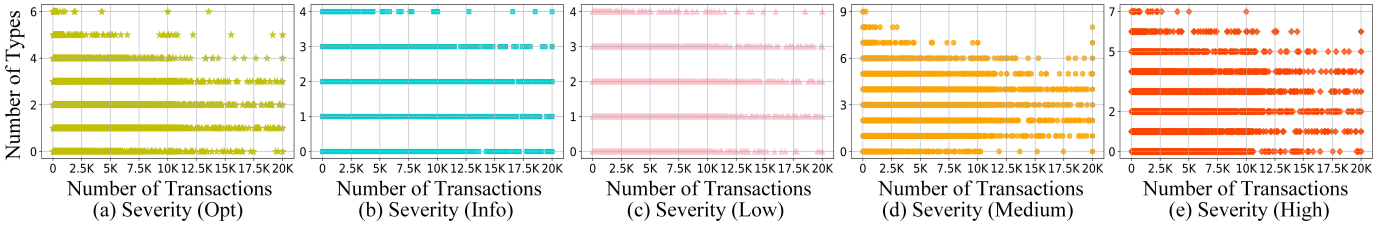


Fig. 6. Correlation between the number of vulnerability types in Dataset_4 detected by VulHunter and the number of contract transactions. In the figure, the upper limit of the number of transactions is set to 20,000 (that is, 20,000 for more than 20,000).

TABLE 10
The comparison of vulnerability number detected on 10,000 bytecodes sampling from Dataset_4.

SE.	Project	VulHunter	Securify	Oyente	Mythril	DefectChecker	SMARTIAN	ContractWard
High	RE	870	741	141	2	215	0	53
	SU	3,815	-	-	22	-	0	0
	CDC	2,033	-	-	0	-	0	0
	Total	15,899	4,010	1,118	24	215	0	92
Medium	LE	1,820	3,127	-	-	402	0	14
	TO	1,799	-	-	940	95	0	0
	UCS	2,037	1,564	-	0	664	1	714
	Total	21,415	5,237	0	1,289	1,846	2	1,742
Low	TS	976	-	298	-	-	0	0
	BP	2,324	-	-	398	282	0	1
	CLL	3,432	-	-	-	123	0	22
	Total	9,915	0	298	398	471	0	62
Info	LLC	3,040	-	-	-	-	-	950
	E20ID	2,050	-	-	-	-	-	0
	HC	2,846	-	-	-	-	-	84
	Total	10,050	0	0	0	0	0	1,034
Opt	ST	3,185	-	-	-	-	-	0
	BE	975	-	-	-	-	-	13
	EF	4,141	-	-	-	-	-	1,273
	Total	14,096	0	0	0	0	0	1,286
Overall	Total	71,375	9,247	1,416	1,711	2,532	2	4,216
	#Failed	0%	14.97%	9.41%	0.02%	10.35%	99.86%	0%
	#Secure	939	3,128	7,948	8,320	7,327	13	6,082

plete but referable. Furthermore, VulHunter can also combine some relatively accurate detection patterns from symbolic execution-based bytecode-level methods (*e.g.*, DefectChecker) to discover misreports, which is discussed in § 5.3.1.

Besides, the harm caused by the contract is related to both vulnerability number and frequency of use. The inactive contracts with vulnerabilities alone may not cause damage to users. Thus, in order to evaluate the contract severity more reasonably, we have introduced the transaction number as the activity frequency of contracts. Fig. 6 presents the correlation between the number of vulnerabilities with various severity and the number of contract transactions. It shows the following phenomenon. (i) A similar distribution held for the different severity, indicating that contracts tend to have vulnerabilities with varying severity. (ii) The number of contract transactions is widely distributed, and most transactions involve vulnerable contracts, suggesting that the contracts on Ethereum can be further improved. (iii) The number of vulnerable contracts gradually decreases as their transactions increase, and this phenomenon is most prominent in *Medium* and *High* severity. This may be attributed to the importance of these contracts prompting the developers to review them in detail. Note that it is dangerous to have many vulnerabilities in active contracts involving numerous transactions, and instead, the harm of vulnerable contracts with fewer transactions may not be serious.

Answer to RQ2. How many vulnerabilities are present in the Ethereum blockchain? Most of the contract source code in Dataset_3 and contract bytecode in Dataset_4 were detected with vulnerabilities ranging from *Low* to *High* severity. Although these vulnerabilities are not prone to exploit and cause harm, identifying them can allow contract owners to focus on risky code and then develop normative contracts. Also, many contracts in these datasets can be optimized to improve their operation status. More importanly, given its detection scalability, performance flexibility, and result verifiability, VulHunter can detect wild contracts better in the future based on the requirements of auditors.

4.4 Execution Overhead of VulHunter (RQ3)

In this section, we present the execution overhead of methods for analyzing Ethereum contracts. First, we selected about 100 contracts with a size of about 121KB, such as the contract with address 0xce5b23f11c486be7f8be4fac3b4ee6372d7ee91e (3,049 lines). Then oscillo [54] was employed to monitor the time and memory overhead of 10 methods for detecting these contracts.As shown in Fig. 7(a), the ML-based (*e.g.*, VulHunter and TMP) and pattern matching-based (*e.g.*, SmartCheck and Slither) methods generally require less time overhead than those based on symbolic execution (*e.g.*, Oyente and Mythril) and fuzzy testing (SMARTIAN). But there are exceptions. For instance, the time overhead of DefectChecker is similar to SmartCheck as its lightweight design. Nevertheless, VulHunter leverages ML to complete the detection within an average of 4.4 seconds. Also, most of time overhead was used to extract instances (3.7s) and load the model (0.65s), which can be improved by parallelism. Notably, although models learn numerous representations for vulnerabilities, they perform fast inference (30 models only require 0.05s), allowing them to be extended to identify more vulnerability types in imperceptible time.

As illustrated in Fig. 7(b), due to thousands of search paths need to be traversed and executed in symbolic execution-based methods such as Oyente, they generally require more memory overhead than ML-based and pattern matching-based methods. Also, the actual running of contracts based on numerous test cases introduces an additional memory overhead in fuzzy testing-based methods. Similar to pattern matching-based methods, the memory overhead of VulHunter mainly consists of three parts: extracted instances, pre-trained models, and intermediate variables during the inference process. The lower memory overhead reflects the main advantage of ML-based methods, *i.e.*, contracts can be accurately analyzed in resource-limited devices. Furthermore,

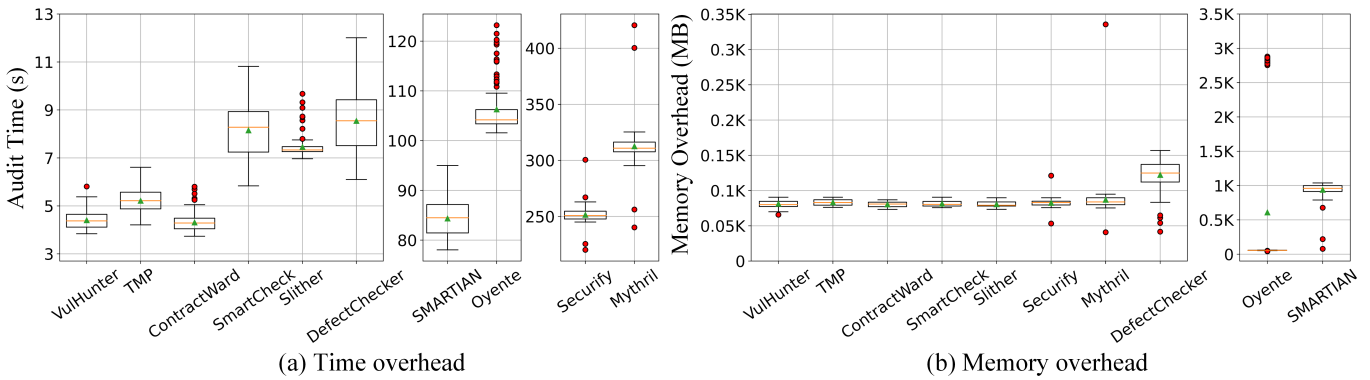


Fig. 7. Comparison of program overhead in terms of time and memory. Each value in the figure refers to the average of 50 execution results.

this phenomenon brings great development potential for VulHunter. For instance, the remaining space can be employed to deploy a private chain that executes vulnerable sequences, thereby ensuring they can trigger the vulnerabilities.

Answer to RQ3. How much overhead does VulHunter require to analyze the smart contracts? VulHunter takes an average of 4.4s and 81.3MB to analyze a 121KB Ethereum contract, which is only one-tenth (or even less) of fuzzy testing-based (*e.g.*, SMARTIAN) and symbolic execution-based (*e.g.*, Oyente) methods. Even more, compared with the pattern matching-based (*e.g.*, Slither) and ML-based (*e.g.*, TMP) methods, VulHunter has superior performance in both time and memory overhead, which delivers the vast potential for future development.

4.5 Authenticity of VulHunter (RQ4)

Towards exploring the superior performance of VulHunter, we demonstrated the detection results of real-world contracts (including those deployed on Ethereum in Dataset_3 and those derived from vulnerability incidents in Dataset_5). Table 11 shows some examples of contracts in Dataset_3.[14]

Detection of *reentrancy-eth* vulnerabilities. VulHunter detected a *reentrancy-eth* vulnerability in the contract *RedExchange* (2.00E+16Wei). The "payFund()" function (corresponding to line 11 of Listing 3) is declared as public. Although this function is guarded with the "onlyAdministrator" modifier, anyone can become a member of administrators by invoking the "RedExchange" function. Moreover, the gas specified by the call function is too large, while the secure gas is usually 2300 ($\ll$40,000). Thus, attackers can construct an attacking contract and utilize the "setBondFundAddress()" function to set the withdrawal address "bondFundAddress" as the attacking contract address, thereby realizing a reentrancy attack. However, methods such as Oyente and DefectChecker missed this vulnerability. Particularly, when "RedExchange" is not provided, the attackers will not be able to invoke "payFund" normally. However, Bi^2-LSTM and other methods such as Slither will still identify this path that includes the reentrant call function as the invariable execution sequence. As detailed in § 5.2.2, it belongs to a false positive of multi-instance collaborations. Notably, VulHunter can build symbolic constraints of vulnerable paths and verify their feasibility with solvers, thus eliminating misreports. Also, this problem can be mitigated by expanding similar contracts in the training dataset, making the model observe these cooperation features. In addition, there are *reentrancy-eth* vulnerabilities in line 1,060 of the *VokenPublicSale* contract and line 77 of the *Acid* contract, which cannot be identified by arts like Oyente, Securify, and Mythril.

14. The details of more examples and the "Others" column are described in https://github.com/ContractAudit/VulHunter/tree/main/Dataset3/Example_results.

```
1  function RedExchange() public {
2    administrators[msg.sender] = true;
3  }
4  modifier onlyAdministrator() {
5    require(administrators[msg.sender]);
6    _;
7  }
8  function setBondFundAddress(address _newBondFundAddress)
       onlyAdministrator() public {
9    bondFundAddress = _newBondFundAddress;
10 }
11 function payFund() payable public onlyAdministrator() {
12   ...;
13   totalEthFundRecieved = SafeMath.add(
         totalEthFundRecieved, ethToPay);
14   if(!bondFundAddress.call.value(_bondEthToPay).gas
         (400000)()) {
15     totalEthFundRecieved = SafeMath.sub(
           totalEthFundRecieved, _bondEthToPay);
16     ...;
17   }
18 }
```

Listing 3. Contract with *reentrancy-eth* (access control with tuples).

```
1  contract Vesting is ... {
2    Token public tokenReward;
3    function createVestingPeriod(..., address
         addressOfTokenUsedAsReward) public {
4      ...
5      tokenReward = Token(addressOfTokenUsedAsReward);
6    }
7    function release(address token) public {
8      ...
9      tokenReward.transfer(_beneficiary, unreleased);
10   }
11 }
```

Listing 4. Contract with *tod*.

Detection of *tod* vulnerabilities. This vulnerability refers to inconsistent behavior caused by miners or nodes interfering with the transaction sequence. It contains three types: key storage variable, owner authentication, and approved tokens. For *Vesting* contract (3.00E+10Wei), the state varible "tokenReward" is assigned in "createVestingPeriod" function and used in "release" function (as shown in Lines 5 and 9 of Listing 4). When the transactions that invoke these two

TABLE 11
Real-world examples of smart contracts on Ethereum. 1st column: detected contract vulnerabilities. 2nd column: contract name. 3rd column: the Ethereum address of the contract. 4th column: number of contract transactions. 5th column: contract balance in Wei (1 ether = 1.00E+18 Wei). 6th column: the line number of the vulnerable code. 7th column: detection results of VulHunter, where "TP" means vulnerability can be detected correctly. 8th column: detection results of other methods, where FNs and FPs represent false negatives and false positives, respectively.

Vulnerabilities	Contract name	Contract address	#Txnums	Balance	Loc	Ours	Others
integer-overflow	WinStar	0x80d0f44bf75ec3e591ac137ac4b88989bb43a006	25	1.60E+15	43	TP	FNs such as TMP
	ETHMaximalist	0xc36fe594db560bfde1bdf5d6b40a7a775d702a1c	78	5.10E+17	171	TP	FNs such as SMARTIAN
	AceTokens	0xf87046beda71bf66c54b74b25ba4b116d93bdb85	143	2.06E+16	255	TP	FNs such as Mythril
	Eightherbank	0xc6e5e9c6f4f3d1667df6086e91637cc7c64a13eb	9,455	1.52E+19	585	TP	FNs such as ContractWard
	AceDapp	0xe65f525ec48c7e95654b9824ecc358454ea9185e	14,202	9.60E+19	242	TP	FNs such as Slither
W/o integer-overflow	DharmaTradeReserve	0x0efb068354c10c070ddd64a0e8eaf8f054df7e26	14,907	9.63E+19	-	TN	FPs such as Oyente
	Trader	0x32c5c27cec2cafef14168451b5a903a2c8917173	29	1.00E+16	-	TN	FPs such as Oyente
	DharmaTradeReserveStaging	0x2040f2f2bb228927235dc24c33e99e3a0a7922c1	1,816	1.23E+18	-	TN	FPs such as Oyente
	DeFi	0x6d72ff6373c86f3b3cca09ff18068d3c8f2b62b6	7	4.00E+15	-	TN	FPs such as TMP
reentrancy-eth	RedExchange	0x5409Fcd56836e0e0459C12Ab45e7Ef23c6094bEd	266	2.00E+16	243	TP	FNs such as Oyente
	VokenPublicSale	0xfeb75b3cc7281b18f2d475a04f1ffaaa3c9a6e36	14	2.90E+15	1,060	TP	FNs such as DefectChecker
	Acid	0x23ea10cc1e6ebdb499d24e45369a35f43627062f	3,136	1.94E+22	77	TP	FNs such as Slither
W/o reentrancy-eth	KlerosGovernor	0x81dcc6246fe261035ffee91cd975faf3d3f3375f	72	0.00E+00	-	TN	FPs such as Slither
	CoinRepublik	0x146645fbc468ad34464240cd494e4f4cd84a120b	158	1.54E+15	-	TN	FPs such as Securify
	Matrix	0xf0542ed44d268c85875b3b84b0e7ce0773e9aeef	6,771	4.42E+19	-	TN	FPs such as Securify
TOD	Vesting	0x38cf1e7839b71171d236acee0bdf292232f7bc97	3	3.00E+10	257	TP	FNs such as TMP
	Marketplace	0x698ff47b84837d3971118a369c570172ee7e54c2	2,346	5.25E+18	78	TP	FNs such as Oyente
	StandardBounties	0x51598ae36102010feca5322098b22dd5b773428b	1,060	2.02E+18	80	TP	FNs such as Securify
locked-ether	SavingAccountProxy	0x7a9E457991352F8feFB90AB1ce7488DF7cDa6ed5	2,348	2.15E+18	339	TP	FNs such as Slither
	Proxy	0x99afe4683d8f6142c2e29b5406db2b88d878cdd1	293	3.86E+15	36	TP	FNs such as DefectChecker
	KyberFeeBurner	0x7702CaaE3D8feE750c4464d80FCb14Ce05e00743	5	1.95E+14	61	TP	FNs such as SMARTIAN
uninitialized-state	Token	0xf6E435b7e8a9fbC1C2B73c9a9FC08aaf65070671	9	1.10E+16	111	TP	FNs such as Slither
	ERC223I	0xaA5bBD5A177A588b9F213505cA3740b444Dbd586	20,000	6.72E+18	86	TP	FNs such as ContractWard
unused-state	Oracle	0x634ab8f3f791a905b6e6ea72c33483401ed56e6b	5,462	5.16E+17	2,979	TP	FNs such as ContractWard
	GSNRecipient	0x2e61c63e045a978b51c6517c79c2592fcfbc82cd	2,912	3.37E+17	1,433	TP	FNs such as TMP
W/o unused-state	Timelocker	0x00000000005330029d3de861454979d0dd8c89dc	1	0.00E+00	-	TN	FPs such as Slither
	Timelocker	0x00000000003709edea9182789f1153e59cfe849e	1	0.00E+00	-	TN	FPs such as Slither
block-other-parameters	Pets	0x540834aa7a6f445b9cd960dd52ea41d382898538	20	2.50E+16	81	TP	FNs such as TMP
	Revolution	0xf3122a43ee86214e04b255ba78c980c43d0073e2	43	4.74E+13	203	TP	FNs such as SmartCheck

functions appear in the same block, the miner can alter the order of transactions to cause an incorrect transfer account. Similarly, the state variable "owner" in the *Marketplace* contract can be assigned twice in a single block, confusing the user sets a different owner. The "StandardBounties" contract is a representative of approved tokens, and the vulnerability is caused by the "erc20-approve" function declared in line 80. When the authorizer changes the authorization, the user creates a consumption transaction that spends the original authorization token and sets more gas than the changed authorization transaction. In this way, the miners will prioritize the consumption transaction, so that the user can spend both old and new authorized amounts. VulHunter successfully identified the above contracts, while methods such as Securify, TMP, and Oyente misreported them. It is noted that this vulnerability also belongs to the cooperation of multi-path executions, which is discussed in § 5.2.2.

```
1  mapping (address => uint256) balances;
2  function transferProxy(uint256 _value, uint256 _feeSmt
       ,...) public ...{
3    if(balances[_from]<_feeSmt+_value) revert();
4    Transfer(_from, msg.sender, _feeSmt);
5    ...
6  }
```

Listing 5. Contract for SmartMesh incident.

In addition, we employ the contracts of well-known vulnerability incidents in Dataset_5 to further clarify the performance of VulHunter. Table 12 describes the information of these contracts, including security incidents, vulnerability names, economic losses, and detection results.[15] **The *integer-overflow* vulnerability incident for the SmartMesh contract.** In April 2018, the transactions of the SmartMesh contract were suspended by various platforms such as Ethereum [55]. However, as shown in Listing 5, attackers can manipulate the input parameter of the "transferProxy" function to make _fee+_value=0 (*integer-overflow*), so that the verification in line 3 will be passed. In this way, attackers can obtain a lot of money. Besides, contracts such as *EMVC* yielded economic losses given the ineffective arithmetic examination performed by the non-conforming SafeMath library. Particularly, although these arithmetic variables are checked in *BEC* contract (as shown in Line 4 of Listing 6), errors in its examination logic also caused an *integer-overflow* vulnerability (line 3).

15. More information such as the detection results is detailed in https://github.com/ContractAudit/VulHunter/tree/main/Dataset5.

TABLE 12
Examples of major smart contract security incidents. Among them, "-" indicates that the information is unknown.

Date	Vulnerabilities	Incidents	Amounts	Loc	Ours	Others
Feb. 2016	DOS	KotET	-	105	TP	FNs such as Oyente
Jun. 2016	reentrancy-eth	The Dao	$60M	201	TP	FNs such as SMARTIAN
Jul. 2017	parity-multisig-bug	Parity bug	$30M	223	TP	FNs such as SmartCheck
Nov. 2017	access permissions	Parity bug 2	$1.52B	111	TP	FNs such as Slither
Feb. 2018	integer-overflow	EMVC	-	16	TP	FNs such as Oyente
Apr. 2018	integer-overflow	BEC	$900M	261	TP	FNs such as Slither
Apr. 2018	integer-overflow	SmartMesh	-	134	TP	FNs such as SmartCheck
Jul. 2018	integer-overflow	AMR	-	205	TP	FNs such as SMARTIN
-	integer-overflow	SIGMA	-	1873	TP	FNs such as SMARTIN
-	integer-overflow	UCN	-	13	TP	FNs such as SMARTIN
-	integer-overflow	ETHX	-	205	TP	FNs such as ContractWard
-	integer-overflow	H3H3	-	205	TP	FNs such as ContractWard
-	integer-overflow	NUMB	-	205	TP	FNs such as ContractWard
-	integer-overflow	POSH	-	183	TP	FNs such as TMP
-	integer-overflow	POWC	-	205	TP	FNs such as TMP
-	integer-overflow	POWH	-	205	TP	FNs such as TMP
-	integer-overflow	POWH	-	205	TP	FNs such as Mythril
-	integer-overflow	POWH3	-	181	TP	FNs such as Mythril
-	integer-overflow	PWHS	-	207	TP	FNs such as Mythril

Thanks to focusing on contract runtime execution sequences and model perception ability to their semantics, VulHunter successfully identified these vulnerable contracts, which were missed by methods such as Slither and SmartCheck.

```
mapping(address => uint256) balances;
function batchTransfer(address[] _receivers, uint256
    _value) ... {
  uint256 amount = uint256(_receivers.length) * _value;
  require(_value>0 && balances[msg.sender]>=amount);
  balances[msg.sender]=balances[msg.sender].sub(amount);
  ...
}
```

Listing 6. Contract for BEC incident.

Answer to RQ4. Can VulHunter discover contracts with substantial and serious vulnerabilities in public chains such as Ethereum? From the above examples, it is concluded that VulHunter can indeed identify contracts that are misreported and underreported by other methods. This illustrates its detection effectiveness on Ethereum contracts. Also, the contract examples in Dataset_3 indirectly verify the authenticity of the detection results in § 4.3. Notably, VulHunter is not only a method for error correction, but also can optimize contracts and reduce unnecessary costs.

4.6 The Performance of Various Baseline Models for Vulnerability Learner (RQ5)

Due to the extensibility of Vulnerability Learner, it can employ various ML models as its engines to make VulHunter embrace different detection abilities, given their distinct characteristics. To illustrate this fact, we ran VulHunter with 11 other supervised ML (*i.e.,* DL and traditional ML) models on contracts in Dataset_1. Note that all models ran under the framework shown in Fig. 5 and used the Bag-instance hybrid attention described in § 3.5. Also, the Bi^2-LSTM and Bi^2-GRU are equipped with the self-model attention based on the Bi-LSTM and Bi-GRU, respectively. Table 13 shows their partial results and total standard deviations.[16] It reflects the following laws. (i) VulHunter is well compatible with multiple supervised models, regardless of DL and traditional ML. It is because all models achieved satisfactory results (*e.g.,* ACC>82%). Note that Graph Neural Network (GNN) can play a better effect in VulHunter than in TMP (81.84%), which also applies to XGBoost (XGB) in ContractWard (70.06%). This reflects the effectiveness of VulHunter in terms of instance information extraction and MIL detection framework. Also, DL models outperform most traditional ML models, as they have great iterative learning, fitting, and generalization capabilities on massive datasets. As shown in Fig. 10, with the training epoch growing, they can gradually understand the data features and steadily improve the metrics, such as accuracy and F1 score. This also demonstrates the framework correctness of VulHunter. Notably, traditional ML models are easy to interpret given

16. The specific results are detailed in https://github.com/ContractAudit/VulHunter/tree/main/Learner_models.

TABLE 13
Comparative results detected by various baseline models for the Vulnerability Learner on Dataset_1 (benign:vulnerable = 2:1).

SE.	Project	Metrics	Bi^2-LSTM	Bi-LSTM	LSTM	Bi^2-GRU	Bi-GRU	GRU	CNN	RF	DT	XGB	SVM	KNN
High	RE	ACC P	95.29 95.06	94.90 95.00	95.69 94.05	96.08 97.47	94.90 92.86	92.94 86.02	**96.47** **100.00**	96.47 98.72	88.63 77.45	96.08 97.47	96.08 98.70	86.67 73.39
		R F1	90.59 92.77	89.41 92.12	92.94 93.49	90.59 93.90	91.76 92.31	**94.12** 89.89	89.41 94.41	90.59 **94.48**	92.94 84.49	90.59 93.90	89.41 93.83	94.12 82.47
	CDC	ACC P	94.87 92.31	87.18 78.57	87.18 75.00	87.18 78.57	89.74 84.62	92.31 91.67	**97.44** **100.00**	94.87 100.00	87.18 75.00	92.31 91.67	89.74 100.00	82.05 71.43
		R F1	92.31 92.31	84.62 81.48	92.31 82.76	84.62 81.48	84.62 84.62	84.62 88.00	**92.31** **96.00**	84.62 91.67	92.31 82.76	84.62 88.00	69.23 81.82	76.92 74.07
	Total NAVG	ACC P	**89.61** 85.35	86.22 78.79	86.34 79.48	86.94 82.24	86.91 81.22	86.88 79.78	88.64 86.02	89.50 **93.10**	83.55 70.44	88.13 89.44	85.55 89.74	81.83 69.54
		R F1	83.43 **84.38**	81.09 79.92	80.22 79.85	77.79 79.95	79.11 80.15	82.21 80.98	79.02 82.37	74.00 82.46	**88.36** 78.38	72.74 80.23	65.22 75.54	82.83 75.60
Medium	LE	ACC P	88.03 80.54	88.23 85.31	88.46 84.80	88.46 89.01	88.07 87.67	88.27 87.25	89.17 86.48	**90.55** 87.33	84.96 72.38	89.84 **95.80**	89.33 90.22	86.50 78.70
		R F1	84.53 82.49	78.16 81.58	79.69 82.17	74.62 81.18	74.73 80.69	75.91 81.19	80.05 83.14	83.83 **85.54**	**88.78** 79.75	72.73 82.68	76.27 82.66	81.58 80.12
	UCL	ACC P	**92.31** **87.50**	88.03 80.49	89.74 84.62	88.89 84.21	88.89 84.21	85.47 76.19	88.03 85.71	90.60 93.75	82.05 67.31	91.45 **96.77**	88.89 96.43	81.20 69.77
		R F1	**89.74** **88.61**	84.62 82.50	84.62 84.62	82.05 83.12	82.05 83.12	82.05 79.01	76.92 81.08	76.92 84.51	89.74 76.92	76.92 85.71	69.23 80.60	76.92 73.17
	Total NAVG	ACC P	**90.78** 88.99	88.84 86.89	88.55 86.13	87.10 85.59	88.50 86.44	88.16 87.45	86.15 83.06	90.54 **94.42**	85.13 74.35	89.09 93.61	87.41 91.01	83.39 75.09
		R F1	86.31 **87.63**	82.48 84.63	82.38 84.21	74.59 79.71	82.13 84.23	79.89 83.50	76.96 79.89	79.23 86.16	**90.65** 81.69	76.11 83.96	72.44 80.67	81.41 78.12
Low	TS	ACC P	89.61 89.26	92.75 98.21	91.30 86.43	91.06 89.15	90.82 84.72	90.34 90.83	90.58 90.24	**93.72** **100.00**	87.44 75.60	93.48 100.00	90.58 94.59	83.09 70.00
		R F1	78.26 83.40	79.71 88.00	87.68 87.05	83.33 86.14	88.41 86.52	78.99 84.50	80.43 85.06	81.16 **89.60**	**92.03** 83.01	80.43 89.16	76.09 84.34	86.23 77.27
	BP	ACC P	**88.14** 86.34	84.03 75.11	84.56 75.49	87.91 85.86	83.88 74.15	83.65 76.48	84.87 84.44	86.54 **91.69**	81.29 67.33	84.94 90.00	82.74 87.81	82.28 70.79
		R F1	76.48 **81.11**	77.85 76.46	79.45 77.42	76.26 80.77	79.22 76.60	73.52 74.97	66.89 74.65	65.53 76.43	**85.16** 75.20	61.64 73.17	55.94 68.34	79.68 74.97
	Total NAVG	ACC P	91.00 88.87	89.60 88.05	89.20 84.08	**91.18** 90.15	89.09 83.32	89.14 86.20	88.96 87.67	89.99 **95.75**	85.78 73.76	89.39 93.91	87.70 91.93	85.67 76.35
		R F1	83.41 86.05	80.50 84.11	83.65 83.87	82.54 **86.18**	84.43 83.87	80.45 83.22	77.75 82.41	73.13 82.93	**89.33** 80.80	72.78 82.00	69.02 78.84	83.51 79.77
Info	LLC	ACC P	90.61 88.31	87.36 88.54	87.29 87.83	89.81 87.80	87.84 82.33	88.16 81.85	87.28 79.39	**90.92** 93.25	87.44 76.69	87.07 96.43	87.64 **96.45**	88.29 83.24
		R F1	82.83 **85.48**	71.33 79.01	71.86 79.05	80.66 84.08	80.90 81.61	82.88 82.36	83.55 81.42	78.45 85.21	**89.56** 82.63	63.59 76.64	65.37 77.92	81.24 82.23
	Total NAVG	ACC P	**90.62** 86.86	87.83 89.04	86.38 82.81	85.49 81.11	86.07 88.74	85.15 85.08	87.81 88.38	89.65 90.80	84.41 74.62	86.33 90.82	87.56 **95.17**	81.98 72.82
		R F1	**85.25** **86.05**	73.22 80.36	76.46 79.51	74.05 77.42	68.77 77.49	69.15 76.29	75.44 81.40	77.21 83.46	81.89 78.08	67.19 77.24	67.01 78.65	77.92 75.29
Opt	ST	ACC P	85.42 82.69	79.89 72.78	79.15 75.00	81.37 78.17	82.84 76.83	83.21 76.47	83.03 79.47	**85.79** **96.43**	76.38 60.82	85.06 94.64	80.44 89.47	77.86 64.59
		R F1	71.27 **82.69**	63.54 67.85	56.35 64.35	61.33 68.73	69.61 73.04	71.82 74.07	66.30 72.29	59.67 73.72	**82.32** 69.95	58.56 72.35	46.96 61.59	74.59 69.23
	BE	ACC P	**92.57** 92.28	89.66 87.52	90.04 85.68	92.04 92.89	90.68 84.71	91.32 88.24	86.19 77.92	87.93 **94.93**	85.36 73.12	86.61 93.18	85.29 92.77	85.06 77.07
		R F1	84.89 **88.43**	80.61 83.92	84.33 85.00	82.53 87.40	88.05 86.35	85.46 86.83	81.96 79.89	67.53 78.92	**88.95** 80.26	64.71 76.38	60.77 73.43	78.80 77.93
	Total NAVG	ACC P	88.38 89.37	84.66 83.88	84.56 85.30	**88.90** 88.61	85.58 84.21	85.10 84.54	84.85 85.42	84.34 94.63	84.39 75.02	84.73 **93.99**	81.34 91.66	77.95 69.48
		R F1	77.82 83.20	74.22 78.75	71.55 77.82	79.63 **83.88**	77.21 80.56	75.95 80.02	73.51 79.01	62.50 75.28	**89.62** 81.68	63.03 75.46	56.08 69.58	71.14 70.30
Overall	Total NAVG	ACC P	**90.04** 87.92 ±1.89 ±2.28	87.36 84.84 ±2.41 ±2.99	87.03 83.69 ±1.72 ±2.39	87.75 85.42 ±1.82 ±2.43	87.30 84.67 ±1.96 ±2.50	86.98 84.60 ±1.56 ±2.05	87.07 85.54 ±2.27 ±2.77	88.87 **93.85** ±1.80 ±1.33	84.60 73.53 ±2.67 ±3.72	87.67 92.38 ±2.23 ±1.86	85.82 91.52 ±**0.00** ±**0.00**	82.06 72.54 ±**0.00** ±**0.00**
		R F1	83.41 **85.60** ±3.17 ±2.99	79.01 81.82 ±4.10 ±3.83	79.09 81.33 ±2.82 ±2.68	77.33 81.18 ±3.64 ±3.22	78.97 81.72 ±3.27 ±3.16	78.29 81.32 ±2.54 ±2.40	76.65 80.85 ±4.14 ±3.73	73.58 82.49 ±4.27 ±3.31	**88.57** 80.35 ±2.05 ±3.09	71.08 80.34 ±5.47 ±4.74	66.30 76.90 ±**0.00** ±**0.00**	79.50 75.86 ±**0.00** ±**0.00**
	#Failed NUMS		0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)

their adequate mathematical foundations, and most of them are so lightweight that they can be trained on small datasets.

(ii) More superior models tend to embrace better results in VulHunter. For example, the performance ranking of DL models is Bi^2-LSTM>Bi-LSTM>LSTM and Bi^2-GRU>Bi-GRU>GRU. This can be attributed to their self-model attention and bidirectional semantic association properties. Also, LSTM-based models are slightly better than GRU-based ones (*e.g.,* Bi-LSTM>Bi-GRU), as LSTM has one more gating unit than GRU, which facilitates it to fit samples. This fact applies to traditional ML models, *e.g.,* Random Forest (RF) outperforms Decision Tree (DT) given its additional bootstrap sampling mechanisms. Therefore, VulHunter can improve its performance by employing superior models in the future.

(iii) Each model has various characteristics and can better detect some vulnerabilities. For instance, DT achieves a high recall rate of 88.57% given its convenience and randomness detection, and RF achieves a high accuracy rate of 93.85% by employing multiple DTs for voting. Moreover, due to the unique model structure and reasoning style, other models deliver more observables on some vulnerabilities. For example, for the *reentrancy-eth* detection, CNN achieved significant accuracy and precision rates (ACC=94.47% and P=100%), and RF obtained the greatest recall rate and F1 score (R=90.59% and $F1$=94.48%). Nevertheless, thanks to modeling capabilities for the front and rear elements in sequences, Bi^2-LSTM holds a superior overall performance, *e.g.,* ACC=90.04% and $F1$=85.60%. More importantly, its self-model attention is the key to locating the defective bytecode fragments and source code statements. In the future, visualization tools such as Captum [56] may help other models obtain the importance distribution of inputs.

(iv) The standard deviation of detection results in VulHunter is mainly related to the model structure/scale. Complex models containing large numbers of neurons have strong generalization/fitting capabilities while also introducing some randomness. For example, Bi-LSTM outperforms LSTM and holds a higher standard deviation. Also, the voting mechanism of RF and attention mechanism of Bi^2-LSTM improve detection performance and reduce prediction fluctuations, *i.e.,* their metrics are superior to those of DT and Bi-LSTM, respectively. Besides, SVM and KNN are held constant during the training process given the certainty of their predictions, obtaining zero standard deviations and inferior performance. Overall, the standard deviation of each model is within an acceptable range, reflecting the detection framework stability of VulHunter.

Answer to RQ5. Can VulHunter support other baseline models for detection? VulHunter can employ multiple DL (*e.g.,* GRU and CNN) and traditional ML (*e.g.,* RF and SVM) models as its detectors and achieve satisfactory performance with acceptable standard deviations. Also, the various models grant it diverse abilities. It can optimize or utilize superior models to improve its detection metrics, and contract auditors can select different models to complete their goals based on time requirements and hardware constraints. For instance, they can combine VulHunter with DT to identify more vulnerabilities, with RF to reduce the workload of manual verification, and with Bi^2-LSTM to make both requirements as compatible as possible.

TABLE 14
Evaluation for instance production hyperparameters.

Setting	n_{block}=32			n_{cycle}=2			n_{seq}=10			**choose**=longest		
Metric	ACC=95.29			P=95.06			R=90.58			F1=92.77		
Time=4.39s				**Memory**=81.27MB								
Parm	n_{block}			n_{cycle}			n_{seq}			**choose**		
Metric	10	16	50	1	5	10	5	20	50	shortest	average	random
ACC	87.45	92.16	96.47	86.67	95.69	96.08	87.84	97.65	97.25	82.35	89.41	89.02
P	84.30	92.79	95.69	85.84	95.61	95.65	87.50	96.52	95.76	80.17	90.00	87.83
R	88.70	89.57	96.52	84.35	94.78	95.65	85.22	96.52	98.26	80.87	86.09	87.83
F1	86.44	91.15	96.10	85.09	95.20	95.65	86.34	96.52	97.00	80.52	88.00	87.83
Time	3.98	4.18	5.02	4.17	4.43	4.44	4.26	4.50	5.12	4.38	4.41	4.40
Memory	76.24	79.24	85.35	79.22	82.15	82.21	76.84	86.47	90.24	79.81	80.84	80.44

TABLE 15
Evaluation for model building hyperparameters.

Setting	T=512		$epoch$=50		λ=0.6		q_b=0.8		q_m=0.2		n_{neurons}=512	
Metric	ACC=95.29		P=95.06		R=90.58		F1=92.77		Time=4.39s		Memory=81.27MB	
Parm	T		$epoch$		λ		q_b		q_m		n_{neurons}	
Metric	256	768	20	100	0.3	0.8	0.6	1.0	0.1	0.3	256	768
ACC	88.63	97.25	91.76	96.86	96.08	93.73	94.12	95.69	95.29	92.55	94.90	95.69
P	86.44	95.76	92.73	96.52	98.17	91.60	91.67	99.06	99.05	89.34	94.74	96.43
R	88.70	98.26	88.70	96.52	93.04	94.78	95.65	91.30	90.43	94.78	93.91	93.91
F1	87.55	97.00	90.67	96.52	95.54	93.16	93.62	95.02	94.55	91.98	94.32	95.15
Time	4.28	5.02	4.38	4.39	4.40	4.40	4.27	4.47	4.30	4.46	4.34	4.48
Memory	78.05	89.01	81.26	81.31	81.28	81.27	80.95	81.41	80.99	81.38	75.11	92.81

4.7 The Effect of Parameters on Performance (RQ6)

To evaluate VulHunter with different hyperparameters, we produce the experiments with varying model settings for detecting *reentrancy-eth* vulnerabilities based on the control variable principle. Table 14 involves the hyperparameter evaluation of instance production. The trends are similar in n_{block} and n_{cycle}. That is, as their values grow, the effective length of instances gradually increases, so that the model can observe more useful information and improve the metrics such as ACC and P. However, since the available instance length for Bi^2-LSTM model is limited to the parameter $T = 512$, the metrics will stabilize when the length exceeds this value. Moreover, the metrics also improve with the number n_{seq} of instances, as more contract execution paths are covered and then facilitate the classifier to make decisions. Meanwhile, the probability of the model outputting malicious (*i.e.,* 1) will also increase, thereby improving the recall rate and reducing the precision rate slightly when n_{seq} overreaches the critical value. For the instance selection strategy, it is more practical to choose the longest instances as the classifier can consider more semantic information. In this way, some vulnerable paths with shorter lengths can be viewed by expanding n_{seq}.

We also test VulHunter with diverse parameters in building the model. As described in Table 15, the metrics are improved in $T = 768$, and reduced in $T = 256$, which can be attributed to the more available information that can be delivered to the classifier. Note that the instances are not as long as possible, as the numerous bytes may dilute valuable information and carry the opposite effect. The parameter $epoch$ holds a similar trend, that is, the model converges gradually as the number of training rounds increases. Nevertheless, excessive rounds may lead to overfitting problems. Moreover, the lower λ makes VulHunter focus on the overall contract labels and more inclined to improve the ACC and P. Instead, it is biased toward detecting individual instances,

so that more vulnerabilities can be discovered but inevitably present some FPs, thereby reducing ACC and P. Given the feasibility verification of the instances, VulHunter can correct some misreports, which is detailed in § 3.7 and § 5.2.2. Besides, q_b and q_m can adjust the number of benign and malicious instances during the iteration process, and the larger q_b can enhance the learning of benign instances. Also, the larger q_m improves the recall rate but reduces the precision rate, due to the benign instances in vulnerable contracts being mislabeled. As discussed in § 5.1.1, they can help VulHunter mitigate the impact of model overfitting and false instance labels. Finally, the number of neurons has an unobvious effect on detection results, and an appropriate value can also alleviate the overfitting problems.

In total, the detection time and memory overhead in these experiments are stable and less device-demanding, which facilitates VulHunter to be deployed in more scenarios. Therefore, serving as a path filter for symbolic executors, VulHunter may help them alleviate the path explosion problem given its accurate and fast inference capabilities. Also, similar to the baseline models, we encourage developers and auditors to allocate appropriate parameters based on their requirements (*e.g.,* biased towards the precision or recall) and hardware limitations (*e.g.,* CPU and memory). In the future, with the maturity of hyperparameter optimization techniques such as scikit-learn [57], they can be considered to tune parameters automatically for better performance.

Answer to RQ6. What is the effect of the variable parameters in VulHunter on its detection performance? Similar to the baseline models, a variety of parameters can also enable VulHunter to meet the various requirements of the contract auditors. For example, working on discovering as many vulnerabilities as possible while tolerating some FPs by improving the value of q_m. Overall, most parameters perform the superior and stable effect, which can guide VulHunter to analyze contract security well.

4.8 Assist in Vulnerability Analysis and Strategy Development at the Source Code and Bytecode Levels (RQ7)

The ultimate goal of the contract inspection is to repair the potential security threat, not just to discover them. However, the current ML-based arts ignore this intention and can only output whether the contracts hold vulnerabilities. To this end, VulHunter identifies vulnerabilities accurately and outputs the specific defective source code statements, key opcodes, and vulnerable execution sequences of contracts, as well as the universal repair methods and delivering remarkable insights for contract developers. In this section, we present two cases on how they can guide the repair of vulnerabilities.

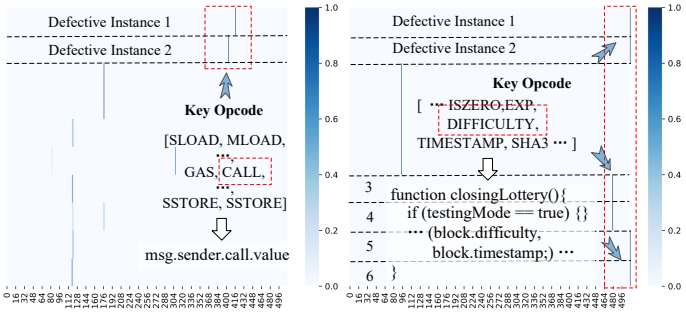
The Dao vulnerability incident. In June 2016, the *reentrancy-eth* vulnerability in the *DAO* contract caused a loss of $60 million ETH. As shown in Listing 7, the behaviors of lines 3-6 describe the implementation of the "withdrawRewardFor" function. Thanks to the unique semantic association and fitting properties, VulHunter identified two sequences with reentrant features from ten contract runtime sequences, while methods like Mythril, SMARTIAN, and TMP cannot. Fig. 8(a) visualizes the attention weight distribution of *reentrancy-eth* detector for instances, and VulHunter obtains the key opcode CALL with weights of 99.94% and 89.55% from two defective instances, respectively. Then it locates the defective statement "msg.sender.call.value" in the contract source code by mapping the ASM file. That is, attackers can deliver the attacks for this statement, and auditors can update this statement to repair this vulnerability.

```
1  function splitDAO(...) ... { ...
2    withdrawRewardFor(msg.sender);
3    // if ((balances[msg.sender] * ratio) / totalSupply <
         paidOut[msg.sender]) throw;
4    // uint reward = (balances[msg.sender] * ratio) /
         totalSupply - paidOut[msg.sender];
5    // msg.sender.call.value(reward)()
6    // paidOut[msg.sender] += reward;
7    totalSupply -= balances[msg.sender];
8    balances[msg.sender] = 0; ...
9  }
10 contract DAOAttack{
11   DAO daoObj;
12   fuction ReAttack() public payable {daoObj.splitDAO()}
13   function() public payable { daoObj.splitDAO()}
14 }
```

Listing 7. The vulnerable code of the *DAO* contract.

Moreover, from the bytecode perspective, auditors or attacks can obtain the defective intersection subsequence [PUSH1 0x14, SLOAD, PUSH1 0x16, SLOAD ... MLOAD ... GAS CALL ... 0x16 SSTORE ... 0x14 SSTORE] around the key opcode, which can convey some useful information about the vulnerability. Among them, 0x14 and 0x16 are the Slot ID of state variables "balances", "totalSupply", respectively. By analyzing these opcodes, the three features of the *reentrancy-eth* vulnerability can be summarized: (i) the GAS consumed by the CALL instruction is not restricted, *i.e.,* there is no specific value of 2300, representing it belongs to a call().value() function; (ii) the "reward" variable loaded by an MLOAD instruction is not a zero constant; (iii) the state variable with the same *Slot ID* is read before invoking the call function and updated after executing the call function.

Similarly, equivalent source code features can be obtained according to the located defective statements. Nonetheless, given that most contracts on Ethereum only own bytecode, observing helpful information at the bytecode level may be slightly hard to understand but more meaningful. Contract analysis tools based on pattern matching, symbolic execution, and fuzzy testing can further use key bytecode or source code to formulate and optimize vulnerability detection rules/logic/oracles. Besides, an attacker can create an attacking contract *DAOAttack* as shown in Listing 7. It can trigger the fallback function by invoking the "ReAttack()" function to execute the "msg.sender.call.value()" statement



(a) *reentrancy-eth* in The Dao (b) *block-other-parameters* in Revolusion

Fig. 8. Case study of locating defective contract source code statements.

in "splitDAO()" again and again, causing three variables to not be updated in time (*i.e.,* lines 6-8 invalid temporarily). Thus, the amount can be continuously withdrawn until the contract balance becomes zero or the GAS is exhausted.

To fix this vulnerability, the developers can advance the position of the SSTORE instruction to before the CALL instruction, preventing attackers from making secondary calls. Also, they are advised to use the "transfer()" function instead of "call.value()" in the source code, given that it can limit 2300 GAS overhead and roll back all transactions on transfer errors. Furthermore, the analysis reports made by VulHunter hold additional information such as vulnerability description, location, key bytecode distribution, and examples with generic repair methods[17], so as to help them develop customized repair strategies more conveniently.

```
1  function closingLottery(...) ... {
2    if (testingMode == true) { return true; }
3    uint randomHash = uint(keccak256(abi.encodePacked(
         block.difficulty,block.timestamp)));
4    uint million = 1000000;
5    uint randomInt = randomHash % million;
6    Trial storage trial = trials[_citizen];
7    uint blocksSince = block.number - trial.
         lastClosingAttemptBlock;
8    if (blocksSince < distributionBlockPeriod) {
9      randomInt *= blocksSince / distributionBlockPeriod;
10   } ...
11 }
```

Listing 8. Contract with *block-other-parameters*.

Detection of *block-other-parameters* vulnerabilities. The contract Revolution (4.74E+13Wei) is an active and wiled contract on Ethereum (as described in Table 11). The part of the source code is shown in Listing 8. As shown in Fig. 8(b), VulHunter identified six execution sequences with the *block-other-parameters* features, and it accurately found that the block parameter variables are used in the "closingLottery()" function based on the key opcodes with weights [84.28%,84.28%,3.94%,3.94%,84.28%,84.28%]. Also, the common sequence around key opcodes is [ISZERO, EXP, DIFFICULTY, TIMESTAMP, SHA3 ... MOD ... PUSH1 0x8, SLOAD, PUSH1 0x7, SLOAD, NUMBER, SUB ... LT, ISZERO ... DIV, MUL]. Among them, 0x8 and 0x7 represent the address of storage variables "trials" and "trial", respectively. Both of them need to be read from the storage using the SLOAD instruction. Also, block-related instructions such as DIFFICULTY and NUMBER are employed to generate random numbers. Since these variables can be known in advance by miners and nodes, the random numbers can be further inferred, thus destroying the fairness of the contract and losing its balance. Therefore, similar to the above case, VulHunter will report these defect positions in the source code and bytecode, and recommend developers to (i) use the business data as the seed of random number generators; (ii) select a combination of multiple pseudo-random data; (iii) employ the online or offline random oracles.

In addition, VulHunter also detected the vulnerability in the *Pets* contract (2.50E+16Wei), while methods such as SmartCheck and TMP missed it. The contract uses block.number to generate the random numbers in the "getRandomNumber" function (line 81), which can also be predicted in advance by working with the miners and nodes. To this end, the contract owners need to avoid these problems during the contract development. Notably, thanks to focusing on the contract execution paths (*i.e.,* opcode sequence or instances), VulHunter can employ the constraint-solving module to verify the feasibility of the vulnerable paths detected by models, thereby eliminating some false positives automatically. The details are illustrated in § 4.9, and more applications of VulHunter are discussed in § 5.4.

17. Refer to https://github.com/ContractAudit/VulHunter/tree/main/Vulnerability_examples/VulnerabilityDescription.xlsx.

TABLE 16
The verification experiments for contracts with infeasible paths. Among them, the symbol "→" refers to the constraint-solving operation.

Contract name (*w/wo* Vul)	VulHunter	Slither	DefectChecker	SMARTIAN	TMP	ContractWard
CIN (*w/* AS)	✓→✓	✓	-	×	×	×
CIN_IFalse (*w/o* AS)	×→✓	×	-	✓	✓	✓
CIN_ITrue (*w/* AS)	✓→✓	✓	-	×	×	×
CIN_IOFalse (*w/o* AS)	×→✓	×	-	✓	✓	✓
CIN_IOTrue (*w/* AS)	✓→✓	✓	-	×	×	×
CIN_AFalse (*w/o* AS)	×→✓	×	-	✓	✓	✓
CIN_ATrue (*w/* AS)	✓→✓	✓	-	×	×	×
CIN_AOFalse (*w/o* AS)	×→✓	×	-	✓	✓	✓
CIN_AOTrue (*w/* AS)	✓→✓	✓	-	×	×	×
APR (*w/* RE)	✓→✓	✓	×	×	×	×
APR_IFalse (*w/o* RE)	×→✓	×	✓	✓	✓	✓
Agent (*w/* IO)	✓→Input	-	-	×	×	×
Cloud (*w/* BP)	✓→✓	-	✓	×	×	×
Cloud_AFalse (*w/o* BP)	×→✓	-	×	✓	✓	✓
Caller (*w/* LLC)	✓→✓	✓	-	-	×	×
Caller_AOFalse (*w/o* LLC)	×→✓	×	-	-	✓	✓
Aether (*w/* ST)	✓→✓	-	-	-	×	×
AEther_IFalse (*w/o* ST)	×→✓	-	-	-	✓	✓

Answer to RQ7. What are the advantages of VulHunter over other ML-based methods in vulnerability repair? Compared with the existing ML-based SOTA arts, VulHunter delivers various services. For instance, it can complete the contract analysis accurately while giving the specific defective source code statements and key opcodes, as well as the universal repair methods. This information can enable developers to check and understand the occurrence mechanisms of vulnerabilities, which can guide them in formulating repair strategies. Also, VulHunter outputs the vulnerable execution sequences, which can be used to build symbolic constraints and compute inputs to eliminate some false positives and restore defective-feasible execution paths, thereby assisting contract auditors better.

4.9 The Capability of Constraint-solving Module (RQ8)

As one of the main advantages of VulHunter, the symbolic constraints of vulnerable paths can be constructed and further solved by SMT solvers to verify their feasibility, like symbolic execution-based methods such as Oyente [4] and Manticore [24]. As detailed in § 3.7, we have initially designed and implemented a constraint-solving module. In order to evaluate the performance of this module in contract vulnerability detection, we analyzed some benign and malicious contract examples holding feasible/infeasible paths. The experiment results are elaborated in Table 16[18]. They reflect the following facts. (i) The static analysis methods, such as pattern matching (*e.g.,* Slither and SmartCheck) and pattern analysis (*e.g.,* Securify), are insufficient to reason the variable states during the contract execution, resulting in some false positives, such as the *arbitrary-send* detection for contracts related to *CIN*. Particularly, due to DefectChecker

18. Detection results of all tools are available at https://github.com/ContractAudit/VulHunter/tree/main/Verification/Infeasible_paths.

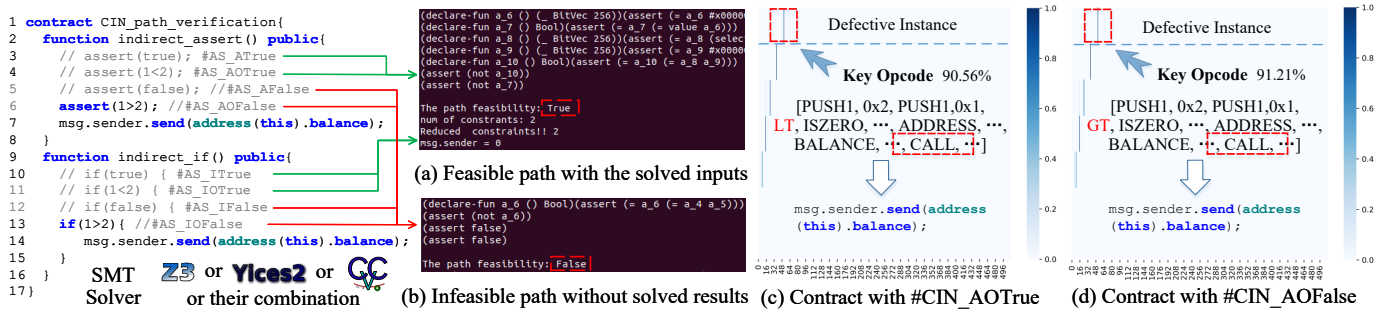


Fig. 9. The feasibility verification for vulnerable paths of contract examples with and without *arbitrary-send*. Among them, each source code comment corresponds to the contracts in Table 16, respectively.

ignoring the constraint-solving in its implementation, it also delivers false positives, *e.g.*, the *block-other-parameters* detection for *Cloud_AFalse* contract.

To this end, we employ the constraint-solving module to solve the constraints of vulnerable instances and correct the misreports. Note that, three solvers, *i.e.*, Z3, Yices, and CVC4, are used one-by-one to illustrate the fact that they are available and effective in the module. As verification examples shown in Fig. 9, the *CIN* contract is embedded with the breaking operations such as if-false and assert-false to construct the feasible and infeasible vulnerable paths. This makes the contracts with/without *arbitrary-send* vulnerabilities. Fig. 9(c) and 9(d) visualize the attention weight distribution of *arbitrary-send* detector for *CIN_AOTrue* and *CIN_AOFalse* contracts, respectively. VulHunter located the suspicious statement "msg.sender.send()" with the confidence of above 90% in both contracts, given almost identical paths with only one different opcode LT (Less-than) and GT (Greater-than). However, the latter vulnerable path is terminated at the assert function, so that the send operation cannot be executed, which belongs to a false positive. The static analysis methods like Slither also misreport it, as they ignore whether the paths can be performed normally. In order to eliminate the false discoveries, VulHunter constructed and solved the constraints of vulnerable paths. Three solvers obtained the same verification results, which is shown in Fig. 9(a)~(b). Among them, the "True" conclusion and inputs that satisfy the constraints were outputted in the case of feasible paths. In contrast, the "False" decision was obtained.

(ii) The imperfect manual detection rules or oracles may make the traditional formal methods miss some vulnerabilities. For example, SMARTIAN ignores the most vulnerabilities such as *arbitrary-send* and *integer-overflow*. This fact is applied to the ML-based methods given their coarse-grained observation. Furthermore, developing the complement detection logic of vulnerabilities requires massive efforts, making the traditional formal analysis methods challenging to detect more types of vulnerabilities. For instance, DefectChecker cannot check the defects such as *send-transfer* and *low-level-calls*. On the contrary, VulHunter leverages ML technology to automatically learn the subtle detection logic and fully exploit information from existing contract datasets, thereby enabling AI-assisted contract analysis.

(iii) The constraint-solving module makes methods such as VulHunter and Oyente hold more capabilities, *e.g.*, obtaining the inputs that restore the vulnerable execution paths. For instance, VulHunter can invoke the "bad" function of *Agent* contract with the parameters solved by *integer-overflow* constraints to cause numerical overflow and get unexpected results[19]. This is conducive to verifying the correctness of detection results and cannot be implemented by the current methods based on static analysis and ML. Also, the ML-based methods such as TMP and ContractWard missed almost defects, as they are insensitive to the subtle features of vulnerabilities. Inspired by the performance of VulHunter, they are suggested to adjust the processed inputs and adopt well-designed models based on mechanisms such as attention, thus improving their detection capability.

In total, this module has implemented the constraint-solving and feasibility verification for the single path at the bytecode/opcode level. Nevertheless, parameter solving for multi-path execution has not been completed, as discussed in § 5.2.2. Also, the challenges of solvers and usable solutions are discussed in § 5.3.2. Therefore, this module is currently available as an optional function in VulHunter, given its incomplete functions and the time consumption for large-scale path verification. In the future, it will optimize solvers and build a state pool to maintain the values of storage variables during the continuous runtime path operation, thus further considering the impact of different path executions.

Answer to RQ8. What is the performance or capability of the constraint-solving module in VulHunter? Currently, the constraint-solving module implements the constraint construction, parameter solving, and feasibility verification for each vulnerable instance, while it needs to be perfected for multiple instance executions. Notably, compared with the existing SOTA arts based on static analysis and ML, VulHunter can not only correct false positives automatically, but also solve the invoking parameters to perform defective paths and trigger the vulnerabilities such as *integer-overflow*.

5 Discussion

5.1 The Rationality of VulHunter

5.1.1 The impact of imperfect instance labels

During the training process of VulHunter, it is inevitable to introduce false labels (called label noise) to instances, given

19. More details are illustrated in https://github.com/ContractAudit/VulHunter/tree/main/Verification/Infeasible_paths.

the empirical initialization and optimization operations on their labels. In order to mitigate the impact of label noise, we propose the Bag-instance hybrid attention mechanism for Bi^2-LSTM model. It enables the model to correct labels of instances in misreported contracts, such as turning malicious instances of benign contracts into benign ones. Fig.10 shows the detection performance of VulHunter with and without this mechanism. Among them, Fig. 10(a) depicts their detection accuracy for 30 types of vulnerabilities.[20] It illustrates that the Bag-instance hybrid attention mechanism is indeed useful for VulHunter to correct misreports and improve its detection metrics, as well as standard deviation. Besides, Fig. 10(b)~(f) describe the changes in model loss and metrics during the training process. The model without this mechanism only learns instance-level features, whose training loss drops rapidly and then becomes stable, *i.e.,* fails to converge. This can be attributed to the fact that it introduced more false labels during the training process, which may lead to model overfitting. Correspondingly, its metrics (*e.g.,* accuracy and precision) are increased and then maintained at an inferior value. Meanwhile, since the larger number of benign instances in the step of instance optimization, it may mistake the features of malicious instances as benign ones and hold an insufficient recall rate.

In contrast, with the guidance of this mechanism, the model can adjust its prediction direction based on the contract true labels, so that it can be continuously optimized (*i.e.,* a declining loss) during the iterative optimization process to steadily improve the recall rate and maintain a superior accuracy rate. Therefore, the better performance of VulHunter is beneficial from the Bag-instance hybrid attention. Also, as described in § 4.6, the model self-attention mechanism enhances the recognition effect and can help mitigate label noise. Furthermore, the ratio of benign and malicious contracts in the training dataset (*c.f.,* § 4.2), as well as the model training parameters λ, q_b, and q_m (*c.f.,* § 4.7), can be adjusted to prevent model overfitting.

20. Results of other metrics are detailed in https://github.com/ContractAudit/VulHunter/tree/main/Rationality.

5.1.2 The theoretical analysis of effectiveness

In this section, we illustrate the effectiveness of VulHunter in terms of instance extraction, model detection, and result validation with theoretical analysis, respectively.[21] **Effective information coverage.** To demonstrate the effective coverage of instance information, we develop a theoretical analysis framework (*i.e.,* instance/path recording entropy model) to quantitatively evaluate the information preserved in the process of VulHunter. Specifically, given that the aperiodic irreducible discrete-time Markov chain (DTMC) can model sequence data [58], we leverage it to formalize an instance as a sequence of opcode operations (random variables). Let $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ denote the state diagram of DTMC, where $\mathcal{V}$ is the set of states (*i.e.,* the variable values) and $\mathcal{E}$ denotes the edges. We define $s = |\mathcal{V}|$ as the number of states and use $\mathcal{W} = [w_{ij}]_{s\times s}$ to denote the weight matrix of $\mathcal{G}$. The state transition matrix $P = [P_{ij}]_{s\times s}$ is constructed based on weights, *i.e.,* $P_{ij} = w_{ij}/w_i$. Similar to [58], we assume that the stationary state distribution $\mu = [\mu_i]_s$ ($\mu_i = w_i = \sum_{j=1}^{s} w_{ij}$) of DTMC is a binomial distribution with the parameter $0.1 \leq p \leq 0.9$ to approach Gaussian distribution with low skewness, and the length of instances obeys a geometric distribution with high skewness (the parameter $0.5 \leq q \leq 0.9$).

$$\mu \sim B(s,p) \to \mathcal{N}(sp, sp(1-p)), L \sim G(q) \qquad (19)$$

Furthermore, it adopts three metrics: (i) the amount of information, *i.e.,* the average Shannon entropy obtained by

21. The information modeling analysis, model effectiveness evidence, and formal verification advantages are detailed in https://github.com/ContractAudit/VulHunter/tree/main/Effectiveness.

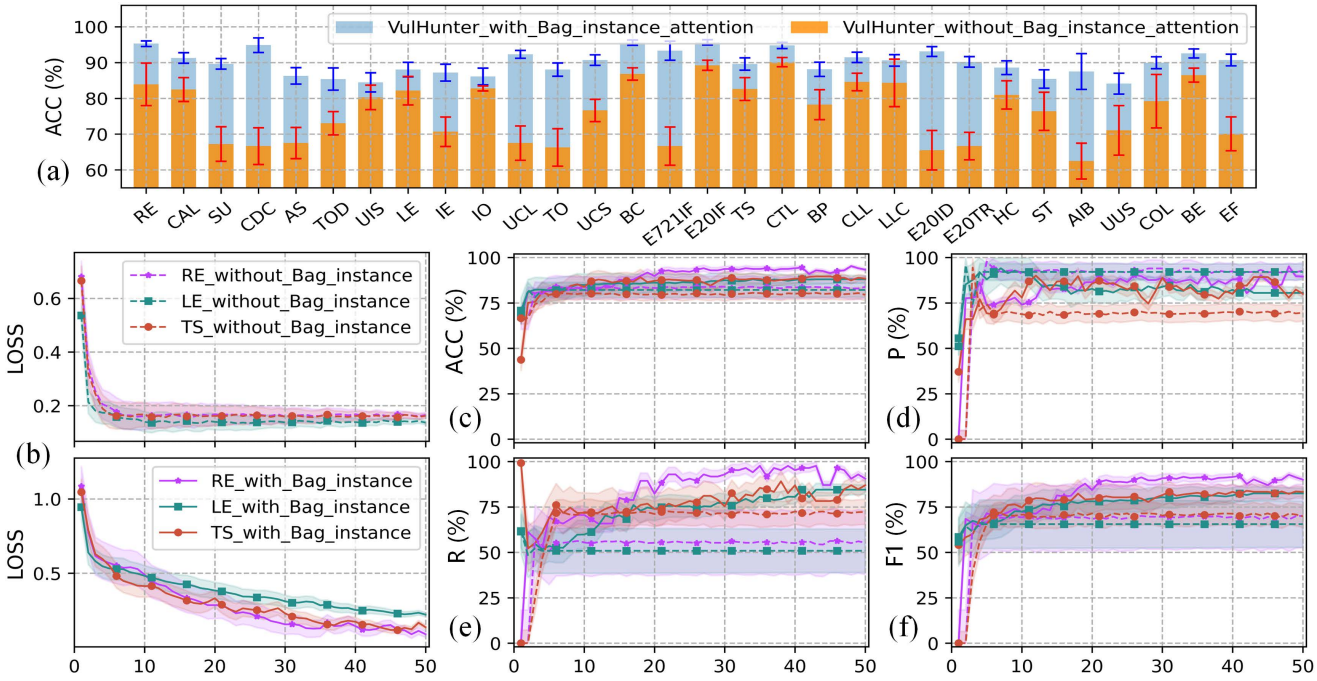
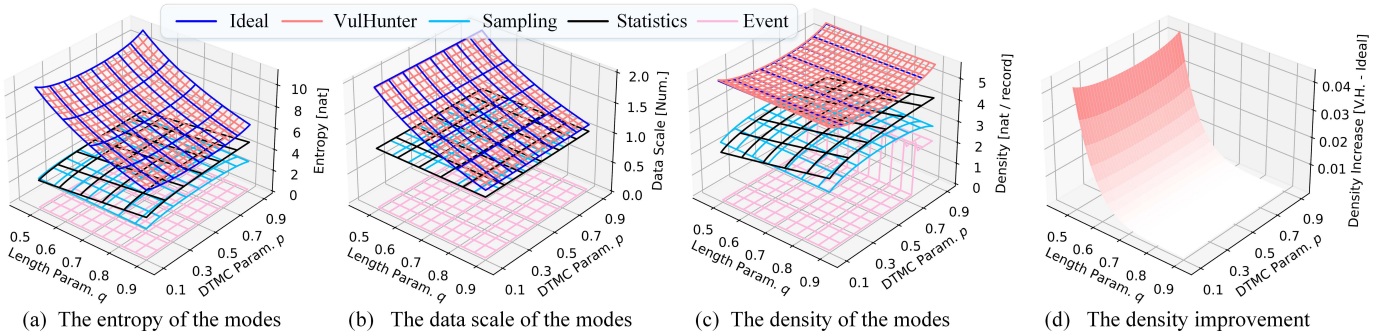


Fig. 10. Performance comparison of VulHunter with and without Bag-instance hybrid attention on Dataset_1 (benign:vulnerable=2:1).



(a) The entropy of the modes (b) The data scale of the modes (c) The density of the modes (d) The density improvement

Fig. 11. The instance/path information retained by different recording modes on the feasible region of the parameters.

recording one opcode operation; (ii) the scale of data, *i.e.*, the space used to store the information; (iii) the density of information, *i.e.*, the amount of information on a unit of storage. By using this framework, we model the opcode sequences-based instance recording mode used by VulHunter, as well as four typical types of recording modes, including (i) idealized mode that records and stores all information of instances; (ii) event-based mode that records specific events for instances denoted by random variable sequences; (iii) sampling/summary-based mode that records coarse-grained instance information, *i.e.*, the sum of opcode values; and (iv) statistics-based mode (*e.g.*, S-gram [13], ContractWard [30], and DeeSCVHunter [17]) that records the statistical state information of instances via several counters.

Then, we select the opcode value as the per-operation feature and perform numerical studies to compare the instance recording modes in real-world settings, *i.e.*, measuring the parameters $|\mathcal{V}|$ and $|\mathcal{E}|$ based on instances of contracts (total of 222,310) in Dataset_1 and Dataset_4. The evaluation results with distribution parameters p and q are shown in Fig. 11, they depict three key facts. (i) VulHunter maintains more information using the opcode sequences of instances than sampling and statistics based recording modes, *e.g.*, it achieves at least 1.5∼3 times information entropy than others. Note that the more long instances (*i.e.*, $q\downarrow$), the more obvious the gap. (ii) VulHunter maintains near-optimal information as its information loss ranging from 5.60×10^{-14} to 6.80×10^{-4} nat. Also, the larger the model hyper-parameter T, the less operation loss. (iii) VulHunter has higher information density than other recording modes, especially for the idealized system. Note that the more long instances, the more obvious superiority. It can be attributed to the fact that VulHunter restricts instance length and reduces the data scale while maintaining as many instance semantics as possible. In summary, VulHunter extracts high-fidelity and compact instance information, which ensures that the model observes enough semantics to identify malicious instance fragments.

Accurate model inference. As tested in § 4.6, VulHunter can employ DL networks (*e.g.*, RNN and CNN) and traditional ML models (*e.g.*, RF and SVM) to detect extracted instances. (i) For the former, their inference effectiveness depends on that of neural networks and hybrid attention mechanisms. Among them, a multi-layer neural network is essentially a composite function, and its fitting ability is demonstrated by the Universal Approximation Theorem, which is similar to the polynomial approximation [59]. On this basis, as detailed in § 3.5, RNN (*e.g.*, LSTM and GRU) connect neurons in the hidden layer through hidden states $\overrightarrow{h_t}/\overleftarrow{h_t}$ and the gating mechanism such as forget gate, enabling it to consider front/back temporal relationships and better handle timing-related tasks such as contract instance analysis. Also, CNN can be viewed as a cascade of linearly weighted filters and non-linear functions for scattering data, and its modeling capabilities were elucidated in [60] from a mathematical perspective.

(ii) VulHunter leverages the self-model attention to improve the perception and fitting ability of the model, enabling it to handle the inputs of long instances without vanishing gradients and overfitting. Also, as discussed in § 5.1.1, the Bag-instance hybrid attention can allow the model to be continuously optimized and learn critical instance features under the guidance of both contract and instance labels.

(iii) The effectiveness of traditional ML models is illustrated by their inherent interpretability, such as adequate mathematical foundations. They can be divided into two categories, namely, mathematical theories-based and rules-based. The former is designed and realized by a series of mathematical operations. For instance, SVM and KNN perform based on the linear/non-linear regression and similarity distance measurement (*e.g.*, Euclidean and Manhattan), respectively. The rules-based algorithms, such as DT, RF, and XGBoost, can be regarded as rule collections in the form of a tree-like structure, which are learned automatically based on contract instance datasets and easy to interpret. Also, they are constructed on the statistical theories, *e.g.*, information gain and Gini index. Besides, more explanations about their characteristics and reasons are illustrated in § 4.6.

(iv) Based on the above reasonable instance analysis by ML models, the MIL framework is used to detect contract defects, whose effectiveness has been demonstrated by its wide spectrum of applications, such as computer vision and natural language processing [61]. Also, some arts [62], [63] combine theorems to analyze the effectiveness of MIL theoretically, and it mainly relies on the premise that a positive bag contains at least one positive instance, whereas a negative bag includes only negative instances. Nevertheless, for contract analysis, this premise retains exceptional cases. That is, against the contract defects caused by multiple instances, the absence of malicious cooperation instances may cause false positives. To this end, as discussed above, the Bag-instance hybrid attention can be utilized to allow the model to observe this nuance, thereby delivering an

accurate decision. Overall, the valuable instance information of contracts can be fitted and represented by the models, and finally make VulHunter identify them under this framework.

Reliable path verification. As experimented in § 4.9 and § 5.2.2, the optional constraint-solving module can validate the feasibility of identified instances and tolerate some false positives of the model. It guarantees the reliability of detection results while enhancing the overall interpretability of our approach, given that the formal analysis method is based on rigorous mathematical foundations (*e.g.*, SMT).

In conclusion, VulHunter holds evidence-based effectiveness in terms of instance information extraction, ML-based model inference, contract overall detection, and result feasibility verification. Notably, the empirical results, including the performance evaluation and example visualization researches in § 4, illustrate the performance superiority of VulHunter, which also can demonstrate the above theoretical analysis and confirm its detection effectiveness.

5.2 The Limitation of VulHunter

5.2.1 Detect vulnerabilities without bytecode-level features

The deep insight of VulHunter is to discover the vulnerable execution sequences of the contract bytecode automatically. However, some vulnerabilities make contracts compile failed, so that their bytecode cannot be generated and VulHunter cannot detect them. For example, contracts with multiple constructors (*multiple-constructors*, *Hign* severity) cannot be compiled. In fact, we can discover them by reviewing the compiler's error messages without using VulHunter.

Moreover, EVM will remove or optimize some semantics (*e.g.*, the compiled version) after the contract compilation. Thus, it is hard to detect the vulnerabilities that depend on this information. For example, incorrect (*solc-version*, *Info*) or multiple (*pragma*, *Info*) developer-specified compiled versions will affect the contract compilation. These defects are suggested to be marked before the contract deployment. However, the contract compilation information is ignored in the bytecode, so as to they are missed. To this end, we must check the pragma [53] in combination with the source codes. Also, the use of the uninitialized state variables (*uninitialized-state*, *High*), storage variables (*uninitialized-storage*, *High*), local variables (*uninitialized-local*, *Medium*), and function pointers (*uninitialized-fptr-cst*, *Low*) will cause unpredictable bugs such as storage slot 0 to be overwritten. It is insufficient to detect them in the bytecode based on whether the contract modifies the value of a specific storage location, as it is unknown that the operation was performed by the vulnerability. As described in § 4.2, we discover the *uninitialized-state* with an inferior ACC of 84.11%. Thus, we need to detect it in the source code by checking whether the variables (*e.g.*, array) are pushed a value before being assigned a storage value.

On the contrary, there are some complex vulnerabilities with many forms that are difficult to be detected by methods such as symbolic execution at the bytecode level, while it is easy to ML techniques. For example, after the execution of critical functions (*events-access*, *Low*) and arithmetic operations (*events-maths*, *Low*), throwing a log event is recommended to notify a caller whether the execution is successful. They can help users track the operation states off-chain and reduce unnecessary errors, as well as wasted gas. However, there are many operations that require adding reminders, such as receiving Ethers. To detect these vulnerabilities, the symbol executors or fuzzers are asked to summarize what kinds of functions need to add reminders, and then detect them one by one. By contrast, VulHunter can detect the vulnerability easily by employing the Bi^2-LSTM model to learn various features of vulnerabilities automatically.

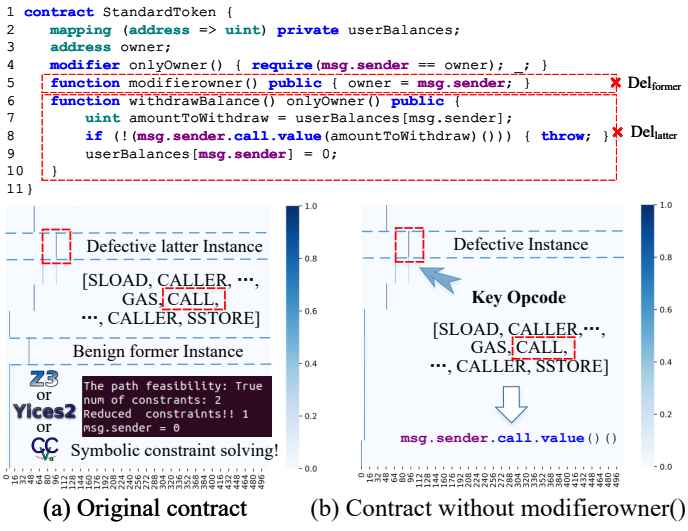
TABLE 17
The results of ablation experiments for detecting vulnerabilities that rely on multi-instance cooperation.

Project	Contract name	Ori.	$\text{Del}_{\text{former}}$	$\text{Del}_{\text{latter}}$	Del_{both}	Other methods
RE	RedExchange	TP	FP	TN	TN	FPs such as Slither
RE	StandardToken	TP	FP	TN	TN	FPs such as DefectChekcer
RE	PullPayment	TP	FP	TN	TN	FPs such as Securify
TOD	Vesting	TP	FP	TN	TN	FNs such as Oyente
AS	AceDapp	TP	FP	TN	TN	FNs such as SMARTIAN
UCL	MyConc	TP	FP	TN	TN	FPs such as SmartCheck

5.2.2 Identify multiple-instance collaboration vulnerabilities

Since the methods such as symbolic execution and ML cannot run the contracts actually, they are challenging to simulate the cooperation of multiple execution sequences (*i.e.*, instances), such as *tod* vulnerability. Nonetheless, VulHunter is doing its best to meet the following observations: (i) For the vulnerabilities triggered by multiple instances, there is a critical instance in the operations triggering the vulnerability, *e.g.*, the last executed instance. In order to explore the key instance which enables VulHunter to identify the vulnerabilities, we performed the ablation experiments. Table 17 shows the detection results of VulHunter and other methods[22], and the first example is detailed in § 4.5. As an another example depicted in Fig. 12, the *StandardToken* contract holds two functions, *i.e.*, "modifierowner()" and "withdrawBalance()". Attackers can first invoke the former function to change the owner of the contract, and bypass the permission of the latter function to execute the reentrancy attack. In fact, the latter

22. The details are illustrated in https://github.com/ContractAudit/VulHunter/tree/main/Verification/Ablation_experiments.



(a) Original contract (b) Contract without modifierowner()

Fig. 12. Contract with *reentrancy-eth* (access control with owners).

function is the key to triggering the vulnerability, and the former is the prerequisite for executing the latter. Specifically, we detected the contract with/without the former and latter functions, respectively. Fig. 12(a) and 12(b) visualize the weight distribution of *reentrancy-eth* detector for contracts with and without the former function. Due to the path invariance, *i.e.,* the deletion of the former instance does not change the latter instance, VulHunter accurately identified the CALL instruction with the 99.96% of confidence and located the defective source code "msg.sender.call.value()" in both contracts. Particularly, VulHunter detected the contract without the latter function as benign. These facts are applied to all examples in Table 17, which reflect that the latter or last instance is most likely the basis for VulHunter identifying multi-instance collaboration vulnerabilities.

As shown in Fig. 12(a), the input solved of vulnerable instance is "msg.sender=0", *i.e.,* the initial value of the storage variable "owner". This can be attributed to the constraint-solving module not implementing the solved parameters associated with the multiple instances at present. In order to discover these false positives for missing cooperation paths, *e.g.,* the detection in Fig. 12(b), we can continue to refine the module to correct them, as detailed in § 4.9. Also, they can be eliminated by checking the defective source code statements or key opcode subsequences based on the tools such as Manticore [24], which are further illustrated in § 5.3.1.

(ii) During the training process, multiple instances of the contract are consecutively trained under the bag-instance attention mechanism to update the state of hidden neurons, thereby realizing the potential connection between numerous instances leading to vulnerabilities. For example, invoking multiple functions (*i.e.,* instances) that operate the same state variables simultaneously (*e.g.,* read and assignment) may cause different results, that is, the *tod* vulnerability. This mechanism can notify the model to adjust its judgments by feedbacking a larger loss when it misses cooperative and malicious instances in vulnerable contracts, so as to make it identify the problems. In this way, multiple malicious instances in some vulnerable contracts can be reported, which may cooperate to trigger vulnerabilities based on their constraint-solved parameters.

5.3 The Improvement of VulHunter

5.3.1 Correcting misreports for executable benign instances

According to the correct rules of symbolic execution-based tools such as DefectChecker [20] that support the source code and bytecode, we can employ them to check the vulnerable bytecode instances. Also, similar to Maian [21], ETHBMC [46] and Manticore [24], VulHunter can build a private chain or simulated executor to create the transaction sequence to execute the suspicious function with the parameters solved by symbolic constraints, and then observe whether the results are unexpected, thus discovering the false positives.

5.3.2 Available optimizations for symbolic constraint solvers

Nowadays, as core components of the constraint-solving module, SMT solvers (*e.g.,* Z3 [42]) are widely used in symbolic execution-based methods such as Oyente [4] and Manticore [24]. Nevertheless, they still need to be improved for the verification of large-scale vulnerable contract paths, *e.g.,* solving complex symbolic constraints with higher efficiency. Although some solvers such as Z3 can handle non-linear and floating-point arithmetic, these unique capabilities introduce additional time overhead, especially for longer constraints. Therefore, more efficient and effective solving algorithms are required for these operations in actual use. The available solutions can be divided into three parts according to their implementation stages.[23] Specifically, (i) pre-processing path constraints before the constraint solving to reduce the constraint complexity, including independent constraint slicing, constraint simplifying, and redundant constraint elimination [64]. (ii) Optimizing solver operations during the solving process, *e.g.,* fast unsatisfiability check, assertion stack optimization, and multi-solver ensemble [45]. (iii) Storing and reusing constraint results after solving, such as constraint storage and incremental solving [65].

5.4 The Application Prospect of VulHunter.

Similar to tools such as Oyente, VulHunter inspects contracts based on source code or bytecode, and report the defective source code statements with multiple vulnerabilities, as well as their opcode subsequences. Also, the symbol constraints of contract execution paths can be built and solved to verify their feasibility, as detailed in § 4.9. As another way of contract protection, run-time monitoring and validation are explored by some methods (*e.g.,* Sereum [66] and Contractguard [27]). They identify and prevent transactions related to vulnerabilities during the contract execution. Similarly, VulHunter can take the execution parameters of contract transactions as inputs, and determine abnormal transactions by verifying them with constructed symbolic constraints for vulnerabilities, guaranteeing the contract operation security.

Meanwhile, as mentioned in § 4.9, it can deliver the contract execution with inputs that meet the constraints of vulnerable instances to restore their paths and trigger the vulnerabilities such as *integer-overflow*. Nevertheless, some vulnerabilities need to execute one path repeatedly or multiple paths simultaneously. For example, the *reentrancy-eth* vulnerability requires repeated execution of paths with the reentrant call function, and its invoking method needs to be selected by implementers based on the vulnerability knowledge, *i.e.,* achieving the reentrant through the auxiliary contract with the fallback function. Therefore, with the refinement of the constraint-solving module, VulHunter can protect the contract security throughout the entire lifecycle from development to deployment in the future.[24]

6 Related Work

Source code based vulnerability detection. Numerous formal verification-based methods have attempted to model the Ethereum contract source code. Bhargavan [67] and Jiao [68] suggest translating a subset of Solidity to F* and K framework for formal verification. Also, Trail of Bits developed a static analysis method called Slither with predefined rules to detect problematic source codes [8]. Similarly,

23. The detailed methods are discussed in https://github.com/ContractAudit/VulHunter/tree/main/Solvers.

24. More improvements and applications are discussed in https://github.com/ContractAudit/VulHunter/tree/main/Discussion.

SmartCheck [7] was proposed by SmartDec and it employs detection rules to determine contract vulnerabilities. Other arts include ZEUS [3] and NeuCheck [14]. However, only <2% of contracts on Ethereum open up their source code [18], [32], [33], which restricts the usage of these methods.

EVM bytecode/opcode based vulnerability detection. The contract bytecodes are visible to everyone, giving an opportunity to evaluate the security of Ethereum contracts. *Theorem proving based methods.* Grishchenko [69] and Hildenbrandt [70] employed F* and K frameworks to transform EVM bytecode in formal tools. Park et al. [71] presented a deductive verification tool to detect the contract bytecode. While these approaches enable formal machine-assisted proofs of various contract security properties, none of them can provide a fully automated analysis. As a result, other automated works based on symbolic execution, etc., have been proposed to ensure the contract correctness and security.

Symbolic execution based methods. Oyente [4], developed by Melonport, builds the CFG from EVM bytecodes and uses pre-defined logical rules to find potential contract problems. On this basis, Osiris [18] improved the detection of integer bugs. Mythril [9], developed by ConsenSys, combines symbolic execution and taint analysis for control flow inspection. Recently, Chen et al. [20] proposed DefectChecker to detect contract defects that can cause unwanted behaviors of Ethereum contracts. Similar methods include ETHBMC [46], Honeybadger [23], VerX [72], and SAILFISH [26].

Fuzzy testing based methods. ContractFuzzer [10] utilizes random fuzzing and pre-defined oracles to find potential contract attack vectors. Subsequently, sFuzz [73] and Echidna [74] combined analysis engines (*e.g.,* TeEher [25]) to detect contract vulnerabilities. Recently, some works [11], [28] have an interest in improving fuzzing. For instance, SMARTIAN [28] generated critical transaction sequences of contracts for the fuzzer with static and dynamic data-flow analysis.

Other methods. Securify [19] and Securify2.0 [75] developed by SRI System Laboratory (ETH Zurich) uses semantic facts and predefined patterns based on EVM bytecode to detect contract vulnerabilities. Wang et al. [27] proposed Contractguard to defend Ethereum contracts against intrusion attacks by matching the benign contract execution paths. In addition, TokenScope [33] defined bytecode rules to detect inconsistent token behaviors with token standards.

In summary, the above methods rely on several expert-defined patterns, rules, or oracles to detect contract vulnerabilities. However, expert rules have the risk of errors, and it is difficult to define bytecode-level patterns that cover complex vulnerabilities completely. Also, as the number of contracts increases rapidly, it is impossible for a few expensive experts to design precise rules by reviewing all contracts.

AI exploration in vulnerability detection. Some arts use ML to analyze contracts without expert knowledge, which can be divided into two categories based on their input types.

Source code based machine learning. Zhuang et al. [12] proposed a temporal message propagation network (TMP) to detect contract source code. Similar methods include AME [49] and CGE [5]. Other arts [13], [15], [16], [17] use ML models such as CNN to learn the characteristics of contract source code and further complete the error detection. These methods are tailored for the source code and cannot analyze numerous contracts with only bytecodes on Ethereum.

Bytecode based machine learning. Huang et al. [29] identified vulnerable contracts by measuring the bytecode vector similarity. Wang et al. [30] proposed a system called Contract-Ward for automated contract vulnerability detection with ML algorithms such as XGBoost and RF. Besides, Hara et al. [76] employed Word2Vec to identify honeypot contract bytecode.

Different from the aforementioned works, on the one hand, VulHunter employs the MIL mechanism and Bi^2-LSTM model to detect contract runtime bytecode paths with contract labels, making it not only discover various vulnerabilities (*e.g., reentrancy-eth* and *timestamp*) in an effective, efficient and interpretable manner, but also identify defective source code statements and vulnerable bytecode sequences. On the other hand, it actively seeks fusion with symbolic execution to build and solve path constraints, thus enabling developers to complete the contract lifecycle more safely.

7 Conclusion and Future Work

We presented VulHunter, a novel ML-assisted detection method for analyzing source code and bytecode/opcode of Ethereum smart contracts without manual pre-defined rules. It leverages the MIL mechanism to address the problem of *classification lacking fine-grained labels*, and employs a self-designed Bi^2-LSTM model to capture the subtle features of benign and malicious contracts for identifying vulnerable instances. Then, it automatically locates the defective source code statements by mapping the key opcodes with the ASM file, and validates their feasibility via SMT solvers. The experiment results on five datasets demonstrate that VulHunter can detect contract vulnerabilities more accurately, efficiently, robustly, and flexibly than SOTA methods. More importantly, compared with ML-based arts, it can provide the defect positions and vulnerable instances while producing classification results, enabling the developers to eliminate the false positives and repair the vulnerabilities more conveniently. In the future, the perfection of the constraint-solve module will make VulHunter embrace bright scenarios, such as contract vulnerability simulation and abnormal monitoring for multiple cooperative transactions.

Acknowledgements

This work was supported by the National Key R&D Program of China (2021YFB2700603), National Natural Science Foundation of China (62172405, 62072487, 62227805, and 62072398), Major Public Welfare Projects Foundation of Henan Province (201300210200), Beijing Natural Science Foundation (M21036), Zhejiang Key R&D Plan (2021C01116), and Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005).

References

[1] T. T. A. Dinh *et al.*, "BLOCKBENCH: A framework for analyzing private blockchains," in *SIGMOD*. ACM, 2017, pp. 1085–1100.
[2] C. Badertscher *et al.*, "Bitcoin as a transaction ledger: A composable treatment," in *CRYPTO (1)*, vol. 10401. Springer, 2017, pp. 324–356.
[3] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *NDSS*. The Internet Society, 2018.
[4] L. Luu, D. Chu, H. Olickel *et al.*, "Making smart contracts smarter," in *CCS*. ACM, 2016, pp. 254–269.

[5] Z. Liu, P. Qian, X. Wang *et al.*, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Trans. Knowl. Data Eng.*, 2021.

[6] Z. Li, S. Lu, R. Zhang, R. Xue, W. Ma, R. Liang, Z. Zhao, and S. Gao, "Smartfast: an accurate and robust formal analysis tool for ethereum smart contracts," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 197, 2022.

[7] S. Tikhomirov *et al.*, "Smartcheck: Static analysis of ethereum smart contracts," in *WETSEB@ICSE*. ACM, 2018, pp. 9–16.

[8] J. Feist, G. Grieco *et al.*, "Slither: a static analysis framework for smart contracts," in *WETSEB@ICSE*. IEEE / ACM, 2019, pp. 8–15.

[9] C. Software, "Security analysis tool for evm bytecode." [EB/OL], https://github.com/ConsenSys/mythril Accessed May 1, 2021.

[10] B. Jiang, Y. Liu *et al.*, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *ASE*. ACM, 2018, pp. 259–269.

[11] J. He, M. Balunovic, N. Ambroladze *et al.*, "Learning to fuzz from symbolic execution with application to smart contracts," in *CCS*. ACM, 2019, pp. 531–548.

[12] Y. Zhuang, Z. Liu *et al.*, "Smart contract vulnerability detection using graph neural network," in *IJCAI*, pp. 3283–3290.

[13] H. Liu *et al.*, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *ASE*. ACM, 2018, pp. 814–819.

[14] N. Lu, B. Wang, Y. Zhang *et al.*, "Neucheck: A more practical ethereum smart contract security analysis tool," *Softw. Pract. Exp.*, vol. 51, no. 10, pp. 2065–2084, 2021.

[15] Z. Gao, V. Jayasundara, L. Jiang *et al.*, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *ICSME*. IEEE, 2019, pp. 394–397.

[16] H. Wu, Z. Zhang, S. Wang *et al.*, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *ISSRE*. IEEE, 2021, pp. 378–389.

[17] X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, "Deescvhunter: A deep learning-based framework for smart contract vulnerability detection," in *IJCNN*. IEEE, 2021, pp. 1–8.

[18] C. F. Torres *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *ACSAC*. ACM, 2018, pp. 664–676.

[19] P. Tsankov, A. M. Dan *et al.*, "Securify: Practical security analysis of smart contracts," in *CCS*. ACM, 2018, pp. 67–82.

[20] J. Chen, X. Xia, D. Lo *et al.*, "DEFECTCHECKER: automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Software Eng.*, 2020.

[21] I. Nikolic, A. Kolluri *et al.*, "Finding the greedy, prodigal, and suicidal contracts at scale," in *ACSAC*. ACM, 2018, pp. 653–663.

[22] N. He, R. Zhang, H. Wang *et al.*, "EOSAFE: security analysis of EOSIO smart contracts," in *USENIX Security Symposium*. USENIX Association, 2021, pp. 1271–1288.

[23] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *USENIX Security Symposium*. USENIX Association, 2019, pp. 1591–1607.

[24] M. Mossberg, F. Manzano, E. Hennenfent *et al.*, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *ASE*. IEEE, 2019, pp. 1186–1189.

[25] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *USENIX Security Symposium*. USENIX Association, 2018, pp. 1317–1333.

[26] P. Bose, D. Das *et al.*, "SAILFISH: vetting smart contract state-inconsistency bugs in seconds," in *IEEE S&P*. IEEE, 2022.

[27] X. Wang, J. He, Z. Xie, G. Zhao, and S. Cheung, "Contractguard: Defend ethereum smart contracts with embedded intrusion detection," *IEEE Trans. Serv. Comput.*, vol. 13, no. 2, pp. 314–328, 2020.

[28] J. Choi, D. Kim, S. Kim *et al.*, "SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *ASE*. IEEE, 2021, pp. 227–239.

[29] J. Huang, S. Han, W. You *et al.*, "Hunting vulnerable smart contracts via graph embedding based bytecode matching," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 2144–2156, 2021.

[30] W. Wang, J. Song, G. Xu *et al.*, "ContractWard: Automated vulnerability detection models for ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, 2021.

[31] Solidity, "Solidity v0.5.0." [EB/OL], https://docs.soliditylang.org/en/v0.5.0/050-breaking-changes.html Accessed May 1, 2021.

[32] Etherscan, "Contracts with verified source codes only." [EB/OL], 2017, https://etherscan.io/contractsVerified Accessed May 1, 2021.

[33] T. Chen, Y. Zhang, Z. Li *et al.*, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *CCS*. ACM, 2019, pp. 1503–1520.

[34] Ethereum, "Solc-js." [EB/OL], 2022, https://github.com/ethereum/solc-js Accessed May 1, 2022.

[35] M. J. Coblenz, "Obsidian: a safer blockchain programming language," in *ICSE-C*. IEEE, 2017, pp. 97–99.

[36] V. Team, "Vyper documentation." [EB/OL], 2020, https://vyper.readthedocs.io/en/latest/ Accessed May 1, 2021.

[37] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[38] Ethereum, "Go-ethereum." [EB/OL], 2022, https://github.com/ethereum/go-ethereum Accessed May 1, 2022.

[39] F. Contro *et al.*, "Ethersolve: Computing an accurate control-flow graph from ethereum bytecode," in *ICPC*. IEEE, 2021, pp. 127–137.

[40] P. Ouyang, S. Yin, and S. Wei, "A fast and power efficient architecture to parallelize LSTM based RNN for cognitive intelligence applications," in *DAC*. ACM, 2017, pp. 63:1–63:6.

[41] C. W. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[42] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, vol. 4963. Springer, 2008, pp. 337–340.

[43] B. Dutertre, "Yices 2.2," in *CAV*, vol. 8559. Springer, 2014, pp. 737–744.

[44] C. W. Barrett, C. L. Conway, M. Deters *et al.*, "CVC4," in *CAV*, vol. 6806. Springer, 2011, pp. 171–177.

[45] H. Palikareva *et al.*, "Multi-solver support in symbolic execution," in *CAV*, vol. 8044. Springer, 2013, pp. 53–68.

[46] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *USENIX Security Symposium*. USENIX Association, 2020, pp. 2757–2774.

[47] PyTorch, "Optim." [EB/OL], 2022, https://pytorch.org/docs/stable/optim.html Accessed January 1, 2023.

[48] PyTorch, "Crossentropyloss," [EB/OL], 2022, https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html.

[49] Z. Liu, P. Qian, X. Wang *et al.*, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in *IJCAI*, pp. 2751–2759.

[50] J. Chen, X. Xia *et al.*, "Defining smart contract defects on ethereum," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 327–345, 2022.

[51] T. Durieux, J. F. Ferreira, R. Abreu *et al.*, "Empirical review of automated analysis tools on 47, 587 ethereum smart contracts," in *ICSE*. ACM, 2020, pp. 530–541.

[52] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *USENIX Security Symposium*. USENIX Association, 2021, pp. 1325–1341.

[53] Ethereum, "The solidity programming language." [EB/OL], 2020, https://github.com/ethereum/solidity Accessed May 1, 2021.

[54] R. Mengnan, "oscillo 1.0.0." [EB/OL], 2019, https://pypi.org/project/oscillo/ Accessed May 1, 2022.

[55] M. Rodler, W. Li, G. O. Karame *et al.*, "Evmpatch: Timely and automated patching of ethereum smart contracts," in *USENIX Security Symposium*. USENIX Association, 2021, pp. 1289–1306.

[56] F. O. Source, "Captum: Model interpretability for pytorch." [EB/OL], 2023, https://captum.ai Accessed January 1, 2023.

[57] SKlearn, "Tuning parameters." [EB/OL], 2023, https://scikit-learn.org/stable/modules/grid_search.html Accessed May 1, 2023.

[58] C. Fu, Q. Li, and K. Xu, "Detecting unknown encrypted malicious traffic in real time via flow interaction graph analysis," in *NDSS*. The Internet Society, 2023.

[59] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991.

[60] S. Mallat, "Understanding deep convolutional networks," *Philos. Trans. Royal Soc. A*, vol. 374, no. 2065, p. 20150203, 2016.

[61] Z. Fu *et al.*, "MILIS: multiple instance learning with instance selection," *IEEE TPAMI*, vol. 33, no. 5, pp. 958–977, 2011.

[62] X. Shi, F. Xing, Y. Xie *et al.*, "Loss-based attention for deep multiple instance learning," in *AAAI*. AAAI Press, 2020, pp. 5742–5749.

[63] M. Ilse *et al.*, "Attention-based deep multiple instance learning," in *ICML*, vol. 80. PMLR, 2018, pp. 2132–2141.

[64] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*. ACM, 2005, pp. 263–272.

[65] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Association, 2008, pp. 209–224.

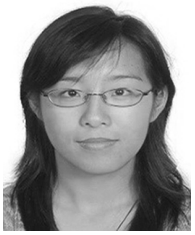
[66] M. Rodler *et al.*, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *NDSS*. The Internet Society, 2019.

[67] K. Bhargavan *et al.*, "Formal verification of smart contracts: Short paper," in *PLAS@CCS*. ACM, 2016, pp. 91–96.

[68] J. Jiao, S. Kan, S. Lin *et al.*, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *IEEE S&P*. IEEE, 2020, pp. 1695–1712.
[69] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *POST*, vol. 10804. Springer, 2018, pp. 243–269.
[70] E. Hildenbrandt, M. Saxena, N. Rodrigues *et al.*, "KEVM: A complete formal semantics of the ethereum virtual machine," in *CSF*. IEEE Computer Society, 2018, pp. 204–217.
[71] D. Park, Y. Zhang *et al.*, "A formal verification tool for ethereum VM bytecode," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 912–915.
[72] A. Permenev, D. Dimitrov, P. Tsankov *et al.*, "Verx: Safety verification of smart contracts," in *IEEE S&P*. IEEE, 2020, pp. 1661–1677.
[73] T. D. Nguyen, L. H. Pham *et al.*, "sfuzz: an efficient adaptive fuzzer for solidity smart contracts," in *ICSE*. ACM, 2020, pp. 778–788.
[74] G. Grieco, W. Song *et al.*, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *ISSTA*. ACM, 2020, pp. 557–560.
[75] E. Z. SRI Lab, "Securify v2.0." [EB/OL], 2020, https://github.com/eth-sri/securify2 Accessed May 1, 2021.
[76] K. Hara, T. Takahashi *et al.*, "Machine-learning approach using solidity bytecode for smart-contract honeypot detection in the ethereum," in *QRS Companion*. IEEE, 2021, pp. 652–659.

Zhaoxuan Li (Student Member, IEEE) is a Ph.D. student in State Key Laboratory of Information Security (SKLOIS), Institute of Information Engineering (IIE), Chinese Academy of Sciences (CAS), Beijing, China. He has published more than 10 papers in international journals and conference proceedings, including TIFS, TDSC, TMC, ESE, COMNETS, and ICWS. His research interests include blockchain security, formal methods, traffic identification, and privacy-preserving.

Siqi Lu is a lecturer and received the Ph.D. degree in Information Engineering University, Zhengzhou, China. He obtained M.Sc. in Cryptography from Information Engineering University, Zhengzhou, China, in 2014. His research interests include formal methods, cryptographic protocol, blockchain, and big data security.



Rui Zhang is an associate researcher with SKLOIS, IIE, CAS, China. She received the Ph.D degree in information security from Beijing Jiaotong University, China, in 2011. She was a postdoctor in Institute of Software, CAS from 2011 to 2013. She was a visiting scholar in Georgia Institute of Technology from 2009 to 2010 and 2018 to 2019. She has published more than 40 technical papers in popular journals and conference proceedings. Her research interests include blockchain security and applied cryptography.

Ziming Zhao is a Ph.D. student in Zhejiang University, Hangzhou, China. He has published more than 5 papers in international journals and conference proceedings, including TIFS, TDSC, TMC, ESE, and AAAI. His research interests include machine learning, traffic identification, and privacy-preserving.

Rujin Liang is an M.D. student in Information Engineering University, Zhengzhou, China. He obtained B.Sc. in Cryptography from Information Engineering University in 2020. His research interests include formal methods and blockchain security.

Rui Xue (Member, IEEE and ACM) is currently a research professor and vice director with the SKLOIS, IIE, CAS. He serves the vice director member of security protocols association in Chinese Association for Cryptologic Research. He has published more than 150 papers in popular journals and international conferences. His research interests include information security and privacy in data and information systems, with a focus on public-key encryption and cryptographic protocols.

Wenhao Li (Student Member, IEEE) received the B.E degree in Computer Science and Technology from Jinan University. He is now pursuing Doctor degree in IIE, CAS, and the School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. He has published more than 6 papers in Network Security in international journals and conference proceedings, including ToN, TIFS, COMNETS, HPCC and ICCS. His research interests include Computer Network Security, Cybersecurity and AI Security.

Fan Zhang (Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the Department of Computer Science and Engineering, University of Connecticut, Mansfield, CT, USA, in 2001, 2004, and 2011, respectively. He is currently a Full Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China, and also with the Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou. His research interests include system security, hardware security, and cryptography.

Sheng Gao (Member, IEEE) is currently an Associate Professor with the School of Information, Central University of Finance and Economics. He received a Ph.D. degree in computer science and technology from Xidian University in 2014. He has published over 30 articles in refereed international journals and conferences. His current research interests include data security, privacy computing, and blockchain technology..