



ByteEye: A smart contract vulnerability detection framework at bytecode level with graph neural networks

Jinni Yang¹ · Shuang Liu³ · Surong Dai¹ · Yaozheng Fang¹ · Kunpeng Xie¹ · Ye Lu²

Received: 24 February 2025 / Accepted: 13 September 2025
© The Author(s) 2025

Abstract

Smart contract vulnerability detection has attracted increasing attention due to billions of economic losses caused by vulnerabilities. Existing smart contract vulnerability detection methods have high false negative and high false positive rates. To address these issues, we present ByteEye, a bytecode level smart contract vulnerability detection framework with Graph Neural Networks (GNNs). ByteEye first constructs an edge-enhanced Control Flow Graph (CFG) to maintain rich information from the low-level bytecode with low latency. ByteEye also designs and incorporates both general information and vulnerability-specific information into its detection method as bytecode level features. Furthermore, ByteEye flexibly supports machine/deep learning models, especially with graph neural networks, which can facilitate vulnerability detection precisely. The extensive experimental results highlight that ByteEye outperforms the state-of-the-art approaches on all three types of vulnerability detection. ByteEye can achieve an average of 35.29%, 43.95%, and 6.38% higher on F1 than the bytecode level best-performed baseline on reentrancy vulnerability, timestamp dependency vulnerability, and integer overflow/underflow vulnerability, respectively. Moreover, ByteEye can detect 361 new vulnerabilities in real-world smart contracts, which are reported for the first time. ByteEye enhances control flow information, designs general bytecode-level features with expert knowledge, and flexibly supports deep learning models, particularly GNNs, thus achieving high detection effectiveness.

Keywords Smart contract vulnerability detection · Graph neural networks · Control flow graph

Extended author information available on the last page of the article

1 Introduction

Smart contracts are program codes deployed on the blockchain system (Yaga et al. 2019; Liu et al. 2023). They enable tamper-resistant and traceable transactions without requiring a trusted third party and thus spawn a wide range of real-world applications (Liu et al. 2023). However, smart contracts may contain vulnerabilities. Exploited vulnerabilities can cause huge economic losses and even threaten the security of the blockchain platform (Durieux et al. 2020; Ghaleb and Pattabiraman 2020; Li et al. 2022). For instance, in the DAO attack (Siegel 2018), the reentrancy vulnerability is exploited and around 4 million *Ethers* (around 60 million USD) are stolen, leading to the hard fork of Ethereum (Wood et al. 2014). In 2020, Binance lost roughly 100 million of cryptocurrency due to the hacking attack (Li et al. 2023).

Smart contract vulnerability detection has attracted increasing attention from both the research and industry communities (Zou et al. 2019). Existing approaches conduct the detection at either the source code level (Luo et al. 2024; Liu et al. 2023; Feist et al. 2019; Xue et al. 2020; Zhuang et al. 2020; Liu et al. 2021) or bytecode level (Luu et al. 2016; ConsenSys 2019; Tsankov et al. 2018; Ma et al. 2022; Liao et al. 2022; Wang et al. 2020; Jiang et al. 2018; Nguyen et al. 2020). Since more than 98% of smart contracts deployed on the blockchain only have their bytecode format released (Torres et al. 2018), the bytecode-based approaches for smart contract vulnerability detection are more applicable and valuable, thus attractive to the latest research (Li et al. 2023; Ma et al. 2022; Liao et al. 2022).

The bytecode level approaches commonly utilize conventional program analysis techniques or exploit the power of machine learning models to detect vulnerabilities. The approaches based on program analysis (Luu et al. 2016; ConsenSys 2019; Tsankov et al. 2018; Ma et al. 2022; Liao et al. 2022) generally convert the bytecode back to higher levels such as control flow graph (CFG) or intermediate representation (Liao et al. 2022; Schneidewind et al. 2020), and they often rely on empirical expert rules. The approaches employing machine learning models (Wang et al. 2020; Huang et al. 2021) mainly consist of analyzing useful information such as statistical information or data flows, extracting and encoding features, and detecting vulnerabilities. However, these approaches usually result in high false negatives and false positives, and alleviating them at the bytecode level is so difficult due to the following three challenges.

Firstly, it is hard to construct an accurate CFG from the bytecode with low latency, such accurate CFG can help capture the high-level syntax information. Mainstream bytecode level approaches (Luu et al. 2016; Ma et al. 2022; Torres et al. 2018; Contro et al. 2021; Krupp and Rossow 2018) employ symbolic execution to construct CFG as the analysis basis, and rely on expert rules to further vulnerability detection. During CFG construction, the unreasonable use of symbolic execution always suffers from path explosion, premature stop, or incomplete code patterns (Chen et al. 2019). It causes the CFG to be incomplete, thus resulting in false negatives. Moreover, symbolic execution is time-consuming (Wang et al. 2020), and it often requires an average of over ten seconds to analyze a single smart contract (Ma et al. 2022), which is detrimental to large-scale smart contracts analysis.

The second is difficulty in extracting precise features from low-level bytecode. The features can reflect smart contract semantics, but CFG lacks explicit information like function and variable names to be extracted. Utilizing machine learning models to design bytecode features usually lacks expert knowledge and protection mechanisms. They just treat bytecode sequences as bigram features or matching objects under inadequate vulnerability-specific knowledge (Wang et al. 2020; Huang et al. 2021), thus also leading to false negatives. Some empirical studies (Durieux et al. 2020; Ghaleb and Pattabiraman 2020; Perez and Livshits 2021) also report that existing approaches show limited consensus on the detected vulnerabilities and have a large number of false positives. Overlooking protection mechanisms used by developers (Xue et al. 2020) will introduce more false positives.

The third is the extensibility to rapidly respond to the fast-evolving new vulnerabilities and new types of vulnerabilities in detection. When coping with new scenarios, approaches employing program analysis require new expert rules and incur higher costs (Liu et al. 2023, 2021), making them less practical. Approaches exploiting machine learning models require continuously updating or integrating various new models (Zhuang et al. 2020; Liu et al. 2021; Wang et al. 2020), even though they have achieved obviously higher recall and precision. Therefore, an extensible smart contract bytecode level framework is so essential for timely and effective new vulnerability detection across the multitude of smart contracts.

To address the above challenges, we present ByteEye, a bytecode level smart contract vulnerability detection framework.

First, ByteEye enhances the edge information to construct the accurate CFG by combining global pattern recognition and local symbolic execution. So it can include sufficient useful edges to maintain richer syntax information from the smart contract bytecode with low latency. Then, ByteEye introduces four types of general information and vulnerability-specific information as bytecode features, which consider both expert knowledge and protection mechanisms to reduce false negatives and false positives. Furthermore, Graph Neural Networks (GNNs) is embedded into ByteEye framework for integrating both local features extracted from bytecode blocks and the global information represented by the edge-enhanced CFG. ByteEye can conveniently support and integrate multiple machine/deep learning models to learn general representations, and helps alleviate the problem of over-reliance on expert rules.

Our extensive experimental results on Ethereum highlight that ByteEye outperforms seven state-of-the-art (SOTA) methods on three types of vulnerability detection tasks over two datasets. In particular, ByteEye performs an average of **35.29%**, **43.95%**, and **6.38%** higher on F1 than SOTA on reentrancy, timestamp dependency, and integer overflow/underflow vulnerabilities, respectively. Our contributions in this paper can be summarized as follows:

- We present an edge-enhanced CFG (E2C) by combining pattern recognition with local symbolic execution. It cost-effectively builds all possible control flow jumps in static analysis from smart contract bytecode. The edge-enhanced CFG is effective in reducing false negatives and achieves an average increment of **11.53%** on recall for all types of vulnerabilities.
- We design multi-dimensional bytecode features considering both general and

vulnerability-specific information. The three types of features including statistics information, spatial structure, and key instructions, contribute to an average of **22.21%**, **12.28%**, and **27.80%** improvement in recall for all types of vulnerabilities. The protection mechanisms feature achieves an average of **8.11%** improvement in precision for all types of vulnerabilities.

- We implement ByteEye as a bytecode level smart contract vulnerability detection framework, and ByteEye outperforms all the compared baseline approaches. We collect more than 33,000 of the most recently released smart contracts in Ethereum. ByteEye can successfully detect vulnerabilities by **10**, **52**, and **299** cases of reentrancy, timestamp dependency, and integer overflow/underflow, respectively, which are reported for the first time.
- We build a new dataset named Smart Contract with Protections (SCP) and manually label it, which has a total of 1145 smart contracts extracted from 47,587 unlabelled smart contracts. In the SCP dataset, 541, 288, and 316 smart contracts relate to reentrancy, timestamp dependency, and integer overflow/underflow vulnerabilities, respectively. We also investigate the well-known ESC dataset with 40,932 smart contracts carefully, revise its pre-process labeling errors, and name it ESC_P.

2 Background and motivation

In this section, background knowledge and our motivations are presented to clarify how we address existing challenges.

2.1 Blockchain and smart contract

Blockchain Blockchains are tamper-resistant digital shared ledgers implemented in a distributed fashion (Yaga et al. 2019). In 2008, Nakamoto presents the modern cryptocurrency Bitcoin as the first application in the blockchain (Nakamoto 2008). Taking smart contracts as execution bytecode, blockchains have various applications such as financial transactions, supply chains, and game development applications.

Smart contract As pieces of program code deployed on the blockchain, smart contracts have two forms, i.e., source code and bytecode. In the famous blockchain platform Ethereum, developers usually use a high-level contract-oriented programming language Solidity (Solidity Programming Language 2023) as the source code. Compiled from source code, the bytecode is stored in each node of Ethereum and executed within the stack-based Ethereum Virtual Machine (EVM) as a series of operation codes (opcodes).

2.2 Vulnerabilities and protection mechanisms

Vulnerabilities Various smart contract vulnerabilities have been reported in the references (Ghaleb and Pattabiraman 2020; Perez and Livshits 2021). We take three influential vulnerabilities as examples to illustrate and evaluate our approach: reen-

trancy (RE) (Ma et al. 2022; Ghaleb and Pattabiraman 2020), timestamp dependency (TD) (Liu et al. 2023, 2021), and integer overflow/underflow (IO) (Torres et al. 2018; Perez and Livshits 2021).

Reentrancy vulnerability¹ always occurs in funds transfers where the attacker utilizes an inconsistent internal state to re-enter the victim contract and steal money many times. Reentrancy vulnerability has not only been the subject of numerous detection studies but also is responsible for substantial economic damages.

Timestamp dependency vulnerability² can be exploited when developers depend on timestamps to make important decisions, such as deciding the game-winner or distributing bonuses. Attackers can change timestamps to manipulate the final result and acquire benefits. An empirical study (Liu et al. 2023) on only 40,932 smart contracts reports that this particular vulnerability impacts 4,833 functions unexpectedly.

Integer overflow/underflow vulnerability³ relates to arithmetic opcodes that have overflow or underflow risks, which can be exploited by attackers to change crucial values or bypass checks, resulting in asset loss. Integer overflow/underflow vulnerability is a prevalent issue in many bytecode application platforms. Detecting this vulnerability at the bytecode level is so challenging due to the absence of explicit function locations and variable names. At the bytecode level, existing detection approaches such as Pluto (Ma et al. 2022) have to rely on the availability of source code to identify this potential vulnerability.

Protection mechanisms To tackle numerous kinds of vulnerabilities and their variants threatening smart contract security, protection mechanisms have been developed to avoid potential attacks. For example, Clairvoyance (Xue et al. 2020) provides a specific protection analysis to the reentrancy vulnerability. We further conclude the protection mechanisms for the three vulnerabilities concerned and provide code examples in Solidity to illustrate corresponding protection forms.

As shown in Table 1, the access control protection mechanism works by only allowing the owner or other reliable accounts to access the crucial but vulnerable program functions, thereby preventing attackers from invading. After a large-scale analysis of real-world smart contracts, we summarize three forms of access control protections, i.e., modifier, keywords and special function, and implicit control access statement. Access control protection gives a comprehensive defense to crucial functions by only allowing trusted accounts. It is a general protection mechanism that works for various vulnerabilities, such as reentrancy vulnerability, timestamp dependency vulnerability, and integer overflow/underflow vulnerability. The protection mechanism is effective in defending against many attacks, yet it restricts the usability of the smart contract.

The check-effect-interaction protection mechanism requires users to reset (e.g., delete, set 0, and decrease amount) the variable that records the balance of *Ethers* before an external call. Using this mechanism, the balance is properly deduced,

¹<https://swcregistry.io/docs/SWC-107>

²<https://swcregistry.io/docs/SWC-116>

³<https://swcregistry.io/docs/SWC-101>

Table 1 Main Protection Mechanisms for Three Vulnerabilities

Protection Mechanisms		Solidity Code Example	Related Vulnerabilities
Access Control	Modifier	onlyOwner	RE
Implicit control access statement	Keywords and special function	{require (owner == msg.sender);} private, internal	TD IO
		function () payable {...} If (msg.sender !=address(this)) throw;	
Check-Effect-Interaction	Delete before call	delete balances[investors];	RE
	Set 0 before call	balances[investors]=0;	
Condition Check Statements	Decrease amount before call	balances[investors]-=amount;	
	Direct condition check	require(...); assert(...); return/throw;	IO
	Indirect condition check	library safemath{...} c=a.mul(b)	

which prevents the attacker from transferring *Ethers* multiple times through a reentrancy call. This protection mechanism is practically effective in classic reentrancy attacks, yet it cannot defend against the cross-function reentrancy attack.

The condition check statements protection mechanisms can effectively defend the integer overflow/underflow attack by directly or indirectly checking crucial arithmetic opcodes. This protection can completely protect the arithmetic opcodes from overflow or underflow.

2.3 CFG for smart contract bytecode

CFG can help capture the control flow information from smart contract bytecode. The node of a CFG is a bytecode instruction block and a directed edge links two blocks where the direction shows the program control transmission (Contro et al. 2021; Chen et al. 2019). The CFG at the smart contract bytecode level is widely used by static analysis approaches (Luu et al. 2016; Ma et al. 2022; Torres et al. 2018; Contro et al. 2021; Krupp and Rossow 2018; Grech et al. 2019, 2018). The most difficult task of constructing a CFG from smart contract bytecode is to build the edges, which has to identify all potential control flow transfers that exist in the bytecode. The main reason is that the jump target of an EVM opcode is stored on the stack rather than directly encoded in the opcode, so the target cannot be obtained by statically parsing opcode.

Figure 1 shows three types of control flow transfers in smart contract bytecode. Type A, Type B, and Type C refer to the final jump addresses that are affected by PUSH opcode, stack-related opcodes, and arithmetic opcodes, respectively. As shown in Fig. 1, Type A shows the simplest format, where the target address is directly available before the jump opcode. The other two types show more complex scenarios where the target address is not directly available before the jump opcode. Type B shows the scenario where the target address (which should be 01B7) is decided by the SWAP opcode. Type C shows the scenario where the target address is the sum of 01 and 01B7. In scenarios of both Type B and Type C, they require calculations of the stack to obtain the target addresses, which usually involves the calculation of multiple blocks.

A recent study (Chen et al. 2019) summarized three widely used static methods to construct CFG for smart contract bytecode. They are pattern recognition, lightweight static analysis methods, and symbolic execution. The pattern recognition method identifies control flow transfers based on deterministic opcode patterns, such as PUSHx/JUMP. Those patterns can only identify simple control flow transfers as shown in Fig. 1-Type A. Lightweight static analysis methods such as reaching definition analysis can identify control flow transfers which the crucial PUSH is set to elsewhere, while they fail to identify Type B and Type C. Symbolic execution can build three types of edges, but it suffers from efficiency overhead, especially for large-scale smart contracts with thousands of program blocks.

2.4 Graph neural networks

With outstanding feature extracting performance, Graph Neural Networks (GNNs) is active in various fields such as node classification, link prediction, and anomaly detection (Luo et al. 2024). GNNs takes the irregular graph as input and is highly successful in handling non-Euclidean data. Graph Convolutional Networks (GCNs) (Kipf and Welling 2017) implements a first-order approximation of spectral graph convolutions and achieves promising performance on semi-supervised classification tasks. Graph Sample and Aggregate (GraphSAGE) (Hamilton et al. 2017) randomly collects neighbors' information to aggregate and update node state. Graph Attention Network (GAT) (Veličković et al. 2018) adds the multi-head attention mechanism to assign different importance to nodes of the same neighborhood. As shown in Fig. 2, the GAT model is utilized to illustrate how GNNs works with the graph input. The GAT model takes the graph as input, which is represented by a feature matrix and an adjacent matrix. The feature matrix will pass through the initial dropout layer and

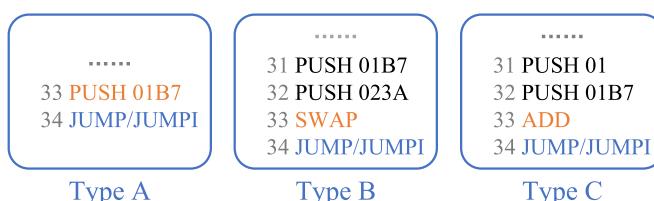


Fig. 1 Three Types of Control Flow Transfers

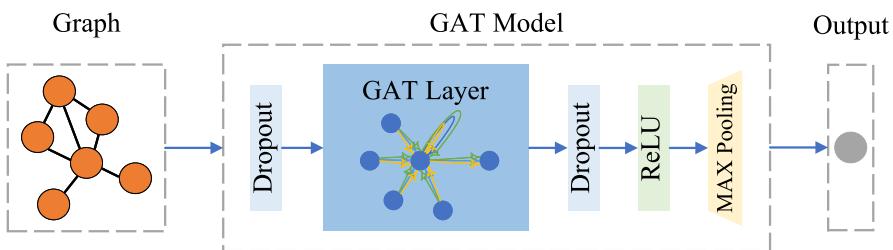


Fig. 2 The Structure of GAT Model

subsequently integrate the multi-head attention mechanism within the GAT layer. The adjacent matrix will help update the values in the feature matrix within the crucial GAT layer and subsequent layers. The ultimate output form of the GAT model is determined by specific tasks.

2.5 Challenges and goals

To clarify our motivation, we use two real examples to explain why existing methods cause a large number of false negatives and false positives, and the issues should be overcome when using GNNs.

Firstly, inaccurate CFGs that lack adequate edge information can cause unreachable blocks and false negatives, while eminently accurate CFGs are always time-consuming. Figure 3(a) reveals a timestamp dependency vulnerability (lines 4-7). In function `getRate` (line 4), changing the timestamp `now` will return different rates. Then the influenced `rate` will affect the computation of the transfer amount `tokens` (line 5), thus affecting the transfer behavior (line 7). Figure 3(b) shows a partial CFG, which corresponds to lines 10-12 in Fig. 3(a). The blocks represent nodes in the CFG and the edges linking the nodes are built based on the opcode in the corresponding block. The pattern recognition method can easily build Edge 1 to Edge 4, with the straightforward pattern that the next address is immediately ahead of the command `JUMP / JUMPI`. For instance, in block 143, the target address is `0cf9`, however, block 149 does not match this pattern and thus pattern recognition method fails to build the edge from block 149 to block 78. Then vulnerability detection based on the constructed CFG will stop at block 149 since this block does not have any edges pointing to other blocks. The edge from block 149 to block 78 corresponds to the return from the function `getRate` in Fig. 3(a). Since the program cannot be successfully executed after `getRate` function, existing methods cannot detect the timestamp dependency vulnerability based on this CFG due to the broken trace. It is worth noting that, using symbolic execution simply and indiscriminately always causes time-consuming in CFG construction, which has been prone to suffer from the path explosion problem (Ma et al. 2022; Chen et al. 2019). Our previous profiling experiments have shown that Oyente takes 3.72 seconds to analyze the contract in Fig. 3(b) and Pluto even costs 72.31 seconds to construct the related CFG. The above reasons motivate us to construct an accurate CFG and reduce time consumption.

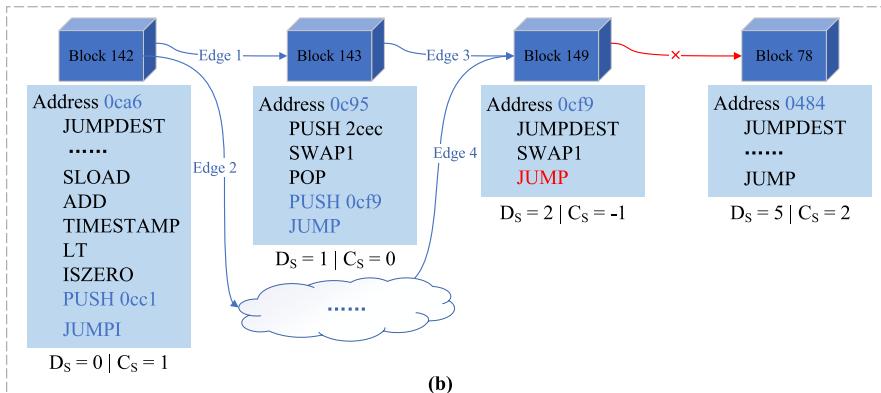
Secondly, mainstream methods even integrate path protection technologies, but incomplete protection mechanisms can only cover a limited range of vulnerability

```

01 contract BitDegreeCrowdsale {
02     function buyTokens(address beneficiary) public payable {
03         uint256 weiAmount = msg.value;
04         uint256 rate = getRate(); x
05         uint256 tokens = weiAmount.mul(rate);
06         .....
07         assert(reward.transferFrom(owner, beneficiary, tokens));
08     }
09     function getRate() internal constant returns (uint256) {
10         if(now < (startTime + 1 weeks)) {
11             return 11500;
12         }
13         .....
14         return 10000;
15     }
}

```

(a)



(b)

Fig. 3 A False Negative Example. This example derives from a source code snippet of the BitDegreeCrowdsale smart contract. The corresponding Ethereum address is 0xa9b2d2be17f3472f9fdb626531f3821080c00159

types, which still leads to an increased risk of false positives. We also provide a false positive motivation example as shown in Fig. 4. Before the crucial call that potentially causes a reentrancy vulnerability (line 10), the smart contract developer employs the access control mechanism (line 4), which only allows the fixed address developer to call the function in line 10. As a result, this does not constitute a reentrancy vulnerability. However, existing approaches (Luu et al. 2016; Tsankov et al. 2018; ConsenSys 2019; Ma et al. 2022) few employ the protection mechanism and will report this smart contract as a reentrancy vulnerability, resulting in a false positive. We have selected 212 safe smart contracts from our SCP datasets and manually checked them. We find that 77.36% of the smart contracts incorporate protection mechanisms. This false positive phenomenon motivates us to consider protection mechanisms for more accurate bytecode feature extraction.

Thirdly, exploiting GNNs to detect vulnerabilities at bytecode level is more challenging than at source code level due to the semantic deficiency gap in the input CFG data, such as the absence of function and variable names. Besides, the CFG used as input to the GNNs must retain extensive information to fully leverage the model's feature extraction capabilities. So these reasons motivate ByteEye to employ GNNs

```

01 contract EnjinBuyer {
02     address public developer = 0x0639C169D9265Ca4B4DEce693764CdA8ea5F3882;
03     function purchase_tokens() {
04         //Protection mechanism: Access Control
05         require(msg.sender == developer); → Protection
06         if (this.balance < eth_minimum) return;
07         if (kill_switch) return;
08         require(sale != 0x0);
09         bought_tokens = true;
10         contract_eth_value = this.balance;
11         //A crucial call statement that may cause a reentrancy vulnerability
12         require(sale.call.value(contract_eth_value)()); → CALL
13         require(this.balance==0);
14     }
15 }
```

Fig. 4 A False Positive Example. The example derives from a source code snippet from the EnjinBuyer smartcontract. The corresponding Ethereum address is 0x6c1bcb34142bffd35f57db626e0ac427af616a4d

and prepare the enhanced CFG at the bytecode level as GNNs input data. ByteEye constructs the CFG to provide GNNs the feature matrix which denotes the characters of each node, and the adjacent matrix which indicates the node connection in the CFG. When processing forward propagation, ByteEye will update the feature matrix according to the adjacent matrix.

3 The ByteEye framework

As shown in Fig. 5, the ByteEye framework consists of three main modules: edge-enhanced CFG construction, feature extraction and encoding, and vulnerability detection. ByteEye first constructs an edge-enhanced CFG from bytecode. Then ByteEye extracts four types of features and encodes two matrices of the edge-enhanced CFG. Finally, ByteEye employs different machine learning models to detect the target vulnerability.

3.1 Edge-enhanced CFG construction

To reduce false negatives, we effectively construct the edge-enhanced CFG (in short E2C) by combining global pattern recognition and local symbolic execution. In Fig. 5, there are two steps to build the edge-enhanced CFG. First, we build basic blocks as nodes of the CFG. Then, we build the edges of Type A by global pattern recognition and build the edges of Type B and Type C by local symbolic execution. The E2C enables the construction of three types of edges.

3.1.1 Build basic blocks

Building basic blocks should pay attention to three aspects. Firstly, for better interpretation and analysis of bytecode semantics, we decode the hexadecimal bytecode into EVM opcodes, which are low-level human-readable references. Secondly, since data segments can not be translated into valid opcodes, we should remove this part

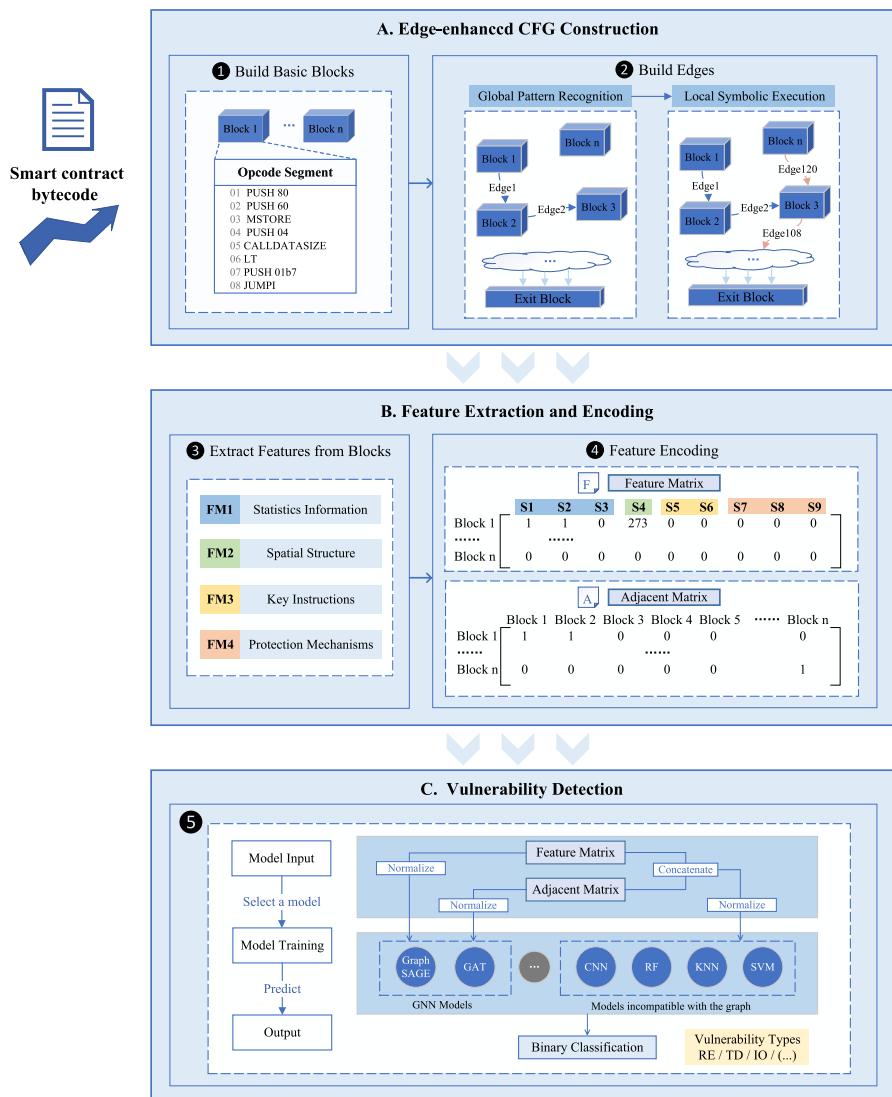


Fig. 5 The ByteEye Framework. ByteEye takes a real-world smart contract as the illustration example. The corresponding bytecode and one target type of vulnerability detection results are viewed as the input and output of the framework, respectively

and meanwhile retain the pure executable opcodes. Thirdly, to obtain the desired blocks, we traverse the pure code segment and split the pure code segment into basic blocks as edge-enhanced CFG nodes. Generally, we view the crucial control flow transfer opcodes as the end flag of a basic block.

3.1.2 Build edges

We combine global pattern recognition with local symbolic execution to build edges of E2C. The detailed process is shown in Algorithm 1. We first build edges of Type A from basic blocks that have fixed patterns using pattern recognition (lines 1-8). Then, we use local symbolic execution to build edges of Type B and Type C in E2C by simulating all stack-related and arithmetic opcodes (lines 9-15). The most critical step is to obtain the block to be simulated in the local symbolic execution, which is implemented in function `getBlocks` (lines 16-29).

Algorithm 1 Build edges of CFG

```

1: Input: block
2: Output: edge
3: for b in block do
4:   e = buildEdge(b)
5:   if e != NULL then
6:     edge.Add(e)
7:   else
8:     blockss.Add(b)
9:   end if
10: end for
11: for b in blockss do
12:   execute_blocks = getBlocks(b, b.Db)
13:   success.e = Symbolic_Execution(execute_blocks)
14:   if success == true then
15:     edge.Add(e)
16:   end if
17: end for
18: Function getBlocks(block b, int s):
19: block_set = getPreviousBlocks(b)
20: if s ≤ 0 then
21:   execute_blocks.Add(b)
22:   return execute_blocks
23: else
24:   for i in block_set do
25:     if recursion depth > limit then
26:       return execute_blocks
27:     end if
28:     getBlocks(i, s - i.Cs)
29:     execute_blocks.Add(b)
30:   end for
31: end if
```

The inputs of Algorithm 1 are the basic blocks, and the outputs are the edges built. To build edges of Type A, we use global pattern recognition, which employs two patterns (PUSH_X/JUMP and PUSH_X/JUMPI) to match the last two opcodes of each block. The function `buildEdge` (line 2) provides the main contents of this method.

When the edge is successfully built, it is added to the returned edge set (line 4). Otherwise, we add the current block into `blockss` (line 6), which contains special blocks that require local symbolic execution to obtain the target address of the edge.

For those special blocks requiring to build edges of Type B and Type C, we execute small-scale opcodes from partial blocks with a minimum symbolic stack rather than execute the whole program to build edges. The function `execute_blocks` (line 11) provides the main contents of this method. In order to simulate the execution with the least number of blocks, we track the preceding blocks of the current block

(line 10) and then conduct symbolic execution on the returned `execute_blocks` (line 11). Once obtaining the execution result successfully, edge building of those special blocks with low overhead (lines 12-15) is finished.

The function `getBlocks` is a recursive function that takes the current specific block b and the minimum required stack depth as input and then returns the minimum sequence of blocks required for the local symbolic execution. The function first obtains all the precedent blocks of the given block b (line 17) and then traces backward following the precedent blocks (lines 21-29). The recursive boundary is decided by two conditions: the minimum stack depth (line 18) and the recursion depth (line 23).

Two variables associated with blocks are defined to associate with calculating the next jump target address. The first variable is the minimum number of elements on the stack (D_s), supporting the successful execution of all opcodes in the current block. The second variable is accountable for describing the change of the stack depth (C_s), which tracks the increased or decreased stack depth after executing the current block. Formulas 1 and 2 calculate D_s and C_s , respectively. We use n to represent the number of opcodes in a block, and a_i and b_i to represent the number of values pushed onto the stack and popped from the stack during executing opcode i , respectively.

$$D_s = \max(b_1, \min\left(\sum_{x=1}^{i-1} (b_x - a_x) + b_i, 1 < i \leq n\right)) \quad (1)$$

$$C_s = \sum_{x=1}^n (-b_x + a_x) \quad (2)$$

We illustrate the process of `getBlocks` with block 149 of the example in Fig. 3(b) in Sec. 2.5. Firstly, we prepare to compute D_s and C_s associated with block 149 based on Formulas (1) and (2). Since `SWAP1` needs two elements in stack and `JUMP` needs one element, D_s of Block 149 is 2, indicating that the minimum stack depth should be 2 before block 149. The `JUMP` opcode pops one element from the stack, thus C_s is -1 for block 149. Then, we use function `getPreviousBlocks` to retrieve all blocks that point to block 149 and collect them into the variable `block_set` (line 17). Additionally, block 143 and some other blocks are not explicitly shown in the example for the sake of simplicity.

The whole recursion process starts at block 149. When executing block 149, we should consider whether block 143 that points to block 149 can provide enough elements or not (line 26). Since C_s of block 143 is 0, we still need two elements in the stack when we are at block 143. We recursively find block 142 that points to block 143, and we need one element in the stack when we are at block 142. The recursion process will stop when (1) the recursion depth exceeds the limit (line 23); (2) `execute_blocks` can provide two elements in the stack at the time of running block 149 (line 18). After executing Algorithm 1, we obtain an edge-enhanced CFG that successfully builds three types of edges with low latency.

3.2 Feature extraction and encoding

To better describe and summarize the vulnerability information, we should design more representative and interpretative features rather than use the original bytecode directly. We first extract the EVM opcode features from the semantics of basic blocks. Next, we generate the feature matrix and the adjacent matrix, which encode the feature information and the edge information of E2C, respectively.

3.2.1 Extract features from blocks

We design four types of features to capture both general information and vulnerability-specific information, including statistics information, spatial structure information, key instructions, and protection mechanisms. Even without explicit function information in bytecode, we can still utilize expert knowledge to analyze bytecode operation semantics and then accurately find the location of the potential vulnerability. Based on the knowledge of protection mechanisms, we can directly discover the related bytecode operations that prevent smart contracts from potential attacks.

Statistics information records the statistics of three specific types of opcodes in each block. The following three types of opcodes can capture the general behaviors of a block (Feng et al. 2016), which helps in classifying similar blocks and distinguishing different blocks. Arithmetic opcodes include ADD, SUB, EXP, and they are represented by two-digit hexadecimal bytecode ranging from 00 to 1d. Memory-related opcodes include MLOAD, MSTORE, SLOAD, and they are represented by two-digit hexadecimal bytecode ranging from 51 to 55. Call-related opcodes include instructions such as CALL, DELEGATECALL, and STATICCALL, and they are represented by two-digit hexadecimal bytecode f1, f4 and fd.

Spatial structure differs from statistics information which mainly captures local information, it captures global information by measuring the importance of the current block. The spatial structure is calculated by the number of offspring a block has. Blocks with many offspring are considered important blocks, as they connect to more blocks and thus pass on more information. We use the spatial structure information to assign weights to a block and blocks with more offspring are more important because they will affect other blocks. The spatial structure tends to indicate the hierarchy layer of a block (Feng et al. 2016), which can help locate similar blocks.

Key instructions are related to specific vulnerabilities, aiming to reduce false negatives in detection. We list the details in Table 2. For reentrancy vulnerability, the key instructions that we are concerned about are CALL and SSTORE, which signal the entrance and the reference of the vulnerability, respectively. For the timestamp dependency vulnerability, the key instructions are CALL and TIMESTAMP together with comparison instruction LT/GT and JUMPI instruction. CALL instruction denotes the fund transfer information reflecting the damage degree of changing the timestamp. The key instructions for integer overflow/underflow vulnerabilities are crucial arithmetic (e.g., ADD), data loading related CALLDATALOAD, and permanent storage-related instructions (e.g., SSLOAD). Since arithmetic instructions at the bytecode level do not only correspond to arithmetic statements in source code, we need more information to find the integer overflow/underflow vulnerabilities. CALLDAT-

Table 2 Key Instructions of Bytecode Features

Key Instructions	Vulnerabilities		
	Reentrancy	Timestamp Dependency	Integer Overflow/underflow
Instruction 1	CALL	CALL	ADD, SUB, MUL
Instruction 2	SSTORE	TIME-STAMP... LT/GT... JUMPI	CAL1DATALOAD
Instruction 3			SSLOAD, SSTORE

ALOAD instruction helps identify arithmetic opcodes, the operands of which are function input variables. Persistent storage-related instructions SSLOAD and SSTORE help find arithmetic opcodes.

Protection mechanisms are considered to reduce false positives in the detection process. To describe protection mechanisms, we utilize three types of information, instructions involved, control flow dependency, and data flow dependency.

Figure 6 illustrates the rules of main protection mechanisms to extract features. Solid arrows represent control dependency and dashed arrows represent data dependency. Figure 6(a) shows the rule to extract feature of the access control protection mechanism. In particular, we require that (1) the block must have CALLER, EQ, and REVERT/JUMPI opcodes; (2) the opcode EQ is control dependent on the opcode CALLER; and (3) EQ is data dependent on CALLER.

Figure 6(b) shows the two rules to extract the features of check-effect-interaction protection mechanism. The first rule represents the checks-effects-interaction pattern that updates (e.g., deletes or sets to 0) the status before an external call. It requires that (1) the block must have PUSH and SSTORE opcodes, and the PUSH opcode must push 0X00 onto the stack. SSTORE describes the operation in that we first acquire

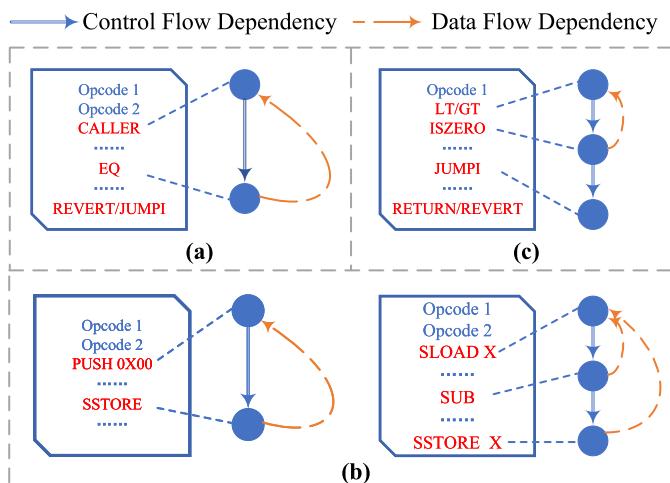


Fig. 6 Rules to Extract Protection Mechanism Features. (a), (b), and (c) relate to the access control, the check-effect-interaction, and the condition check statements protection mechanisms, respectively

two values in the stack as the index and the value, then store the value into storage; (2) the SSTORE opcode is control dependent on the PUSH opcode; and (3) SSTORE is data dependent on PUSH. The second rule represents the pattern that decreases the status variable balance before an external call. It requires that (1) the block must have SLOAD, SUB, and SSTORE opcodes. SUB pops two values from the stack, executes the subtraction, and then pushes the result onto the stack. SLOAD loads the value from permanent storage, and SSTORE stores the value in storage; (2) SUB is control dependent on SLOAD, and SSTORE is control dependent on SUB; and (3) the value acquired from SLOAD must be used in the SUB opcode, and the storage address of SLOAD should maintain consistency with the storage address of SSTORE.

Figure 6(c) relates to the condition check statements protection mechanism. It requires that (1) the block must have the comparison opcode (e.g., LT, GT), ISZERO, and JUMPI in the current block, and REVERT or RETURN in the next block; (2) ISZERO is control dependent on the comparison opcode, and JUMPI is control dependent on ISZERO; (3) the comparison opcode is data dependent on ISZERO.

3.2.2 Feature encoding

We generate a feature matrix that provides initial bytecode feature representations for the blocks in edge-enhanced CFG. For each block, we use a vector of multiple dimensions to store statistics information, spatial structure, key instructions, and protection mechanisms, respectively. The vectors compose the desired feature matrix, and when ByteEye undergoes training, the features of the block are updated as it exchanges information with adjacent blocks.

There are two feature encoding types including statistical counting and bit flags. Statistics information, spatial structure information, and key instructions belong to the first type. For the statistics information feature, we count the number of arithmetic opcodes, memory-related opcodes, and call-related opcodes on a block and encode them as three dimensions in the feature vector. For the spatial structure feature, we count and encode the number of offspring as a dimension. For the key instructions feature, we encode the number of vulnerability-specific opcodes on the block listed in Table 2. The protection mechanism feature belongs to the second type. When the opcodes satisfy the rules in Fig. 6, we assign the corresponding dimension the value of 1, otherwise assign 0.

We also construct an adjacent matrix for encoding edges of the edge-enhanced CFG. This adjacent matrix can help GNNs collect information about neighbor blocks and perform convey or aggregation operations. During ByteEye training, it helps update the mentioned feature vectors. We traverse the edge-enhanced CFG to find the block connected to the other block and record corresponding values in the adjacent matrix as 1.

3.3 Vulnerability detection and framework extension

To perform vulnerability detection, we embed machine learning models into our framework ByteEye. To detect a specific type of vulnerability in a batch of smart contracts, we employ two types of models to conduct detection in ByteEye. These

two types include the state-of-the-art models such as GraphSAGE and GAT, and the widely used models in traditional detection such as Support Vector Machine (SVM) (Suykens and Vandewalle 1999), Random Forest (RF) (Breiman 2001), k-Nearest Neighbor (KNN) (Cover and Hart 1967), and Convolutional Neural Network (CNN) (Lecun et al. 1998). We take the feature matrix and the adjacent matrix as these model inputs.

For GNN models, e.g., GraphSAGE and GAT, we directly utilize two matrices as the standard input. In the model training stage, we continually update the values in the feature matrix indicated by the adjacent matrix. Through multiple layers including graph convolution, dropout, and pooling, we turn the feature matrix into the final binary classification result. When the result value is 1, ByteEye detects the target vulnerability in the current smart contract. For other machine learning models incompatible with the graph structure, e.g., SVM, RF, KNN, and CNN, due to data format alignment, we need to concatenate two matrices into a new matrix with reduced dimensions as a new input. After training, we get the final binary classification values as vulnerability detection results.

We are concerned about two aspects of the extensibility of ByteEye, namely new vulnerabilities and new models. Extending new types of vulnerability detection to ByteEye only needs to update two feature modules in the feature matrix. For a new type of vulnerability introduced, we simply update the key instructions module and the protection mechanisms module. The two modules describe the characteristics of the vulnerability and its corresponding protection mechanisms, respectively. New machine learning models are also easily complemented into ByteEye. As aforementioned, incorporating new GNN models can directly use the two matrices as the standard input.

For the other new machine learning models incompatible with the graph structure, we adjust two matrices into a new matrix and take this as the input.

4 Evaluation

To validate key parts of BYTEEYE and demonstrate its performance benefits, we deploy a set of extensive experiments. We first conduct an ablation experiment to evaluate the effectiveness of three main modules of ByteEye and choose the best performance GNN model. Then we evaluate the overall effectiveness of ByteEye by comparing it with baseline approaches. Furthermore, we validate the detection ability of ByteEye in real-world scenarios. Evaluation experiments aim to answer the following research questions.

- What does each module of ByteEye contribute to vulnerability detection?
- How effective is the ByteEye framework in smart contract vulnerability detection compared with SOTA?
- How many new vulnerabilities can ByteEye detect in real world scenarios?

4.1 Experimental settings

Experimental environment All experiments are conducted on a server equipped with Intel(R) Xeon(R) E5-2678 CPUs (2.5 GHz), three NVIDIA RTX3090 GPUs, and an Ubuntu 20.04.2 LTS Operating System. We implement the ByteEye framework in Python 3.7.13 and adopt PyTorch 1.10.1 to implement different neural network models. To support the open science policy, we make our datasets and source code available at <https://github.com/nkicsl/BYTEEYE>.

Target vulnerability types We adopt three influential vulnerabilities as our targets, including reentrancy, timestamp dependency, and integer overflow/underflow. As aforementioned in Section 2, these vulnerabilities are widely distributed and easily exploited by attackers.

Datasets We use the Smart Contract with Protections (SCP) and the ESC_P dataset for evaluation. Table 3 lists the statistical information of the two datasets, and Fig. 7 displays the distribution of positives and negatives in these datasets.

We manually label our SCP dataset considering protection mechanisms for the three types of vulnerabilities. We collect the raw smart contracts from SmartBugs (Durieux et al. 2020) and label them, which includes a total of 47,587 unlabelled smart contracts. As shown in Fig. 7, our SCP dataset has 541 (80 positives, 461 negatives), 288 (60 positives, 228 negatives), and 316 (50 positives, 266 negatives) smart contracts related to reentrancy, timestamp dependency, and integer overflow/underflow, respectively. The compilation versions of Ethereum smart contracts range from 0.4.11 to 0.6.10. We first utilize several existing vulnerability detection tools to filter the raw smart contracts. Next, we manually annotate the filtered smart contracts following the standard labelling process and check the labels for correctness. To ensure data quality, the creation of the SCP dataset emphasizes the following considerations:

Table 3 Datasets Statistic Information

	Basic Information		Contract Details		Annotation	
	Datasets	Source	Size	Versions		
SCP	SmartBugs (47,587 contracts)	SmartBugs	541 (RE)	0.4.11 - (TD)	285.56 (RE)	Machine Filter +
			288 (TD)	0.6.10	354.07 (TD)	Manually Labelled
			316 (IO)		397.63 (IO)	
ESC_P	ESC (40,932 contracts)	ESC	273 (RE)	None	12.92 (RE)	Preprocess +
			132 (TD)		355.96 (TD)	Manually Checked/ Labbelled
			275 (IO)		13.59 (IO)	

	RE Positives	RE Negatives	TD Positives	TD Negatives	IO Positives	IO Negatives
SCP dataset	461	80	228	60	266	50
ESC _P dataset	200	73	81	51	194	81

Fig. 7 Distribution of Positives and Negatives in Two Datasets

inter-annotator agreement, labeling guidelines, and data quality for protection mechanism recognition.

- Inter-annotator agreement. Two experts independently label the same smart contracts to ensure label correctness. In case of any inconsistent labelling, we refer to vulnerability definitions and classic examples, and conduct discussions until achieving consensus.
- Labeling guidelines. We focus on selecting representative positive samples from real-world scenarios, classic examples, and complex cases capable of bypassing protection mechanisms. The labeling process comprises four key steps: identifying main smart contracts, locating potential vulnerabilities, assessing vulnerabilities, and then checking for applied protection mechanisms to determine final labels.
- Data quality for protection mechanism recognition. We pay attention to samples with protection mechanisms that might be misclassified as vulnerable by automated tools. Each smart contract is meticulously inspected against predefined lists of protection mechanisms (e.g., check-effect-interaction for reentrancy, condition checks for timestamp dependency, direct or indirect condition checks for integer overflow/underflow) to correctly label them as non-vulnerable when adequate safeguards are confirmed. This ensures the dataset accurately reflects cases where vulnerabilities are effectively mitigated.

Although the open source ESC dataset⁴ consists of 40,932 Ethereum smart contracts with around 307,396 functions (Liu et al. 2021; Zhuang et al. 2020), the released data in the dataset only contains pre-processed smart contracts without compilation versions and is neither manually labelled nor directly available. Therefore, we have also manually labelled the pre-processed smart contracts of the ESC dataset, taking into consideration protection mechanisms, and named it as ESC_P dataset. As shown in Fig. 7, the revised dataset ESC_P has 273 (73 positives, 200 negatives), 132 (51 positives, 81 negatives), and 275 (81 positives, 194 negatives) smart contracts related to reentrancy, timestamp dependency, and integer overflow/underflow, respectively.

Measure metrics To measure how ByteEye performs in reducing both false positives and false negatives, we report the general metrics for each evaluated method,

⁴ from <https://github.com/Messi-Q/Smart-Contract-Dataset>.

including Accuracy, Precision, Recall, and F1 score which is a comprehensive evaluation metric that balances Precision and Recall. Additionally, to evaluate the actual detection capability of each method, we define the metric *success rate*. It refers to the percentage of correctly predicted samples within the dataset, determined by dividing the number of successfully predicted samples by the total number of samples. *success rate* provides a reference for the applicability of methods on a given dataset.

Compared baselines We compare ByteEye with the seven baseline approaches, including four bytecode level analysis tools and three source code level analysis tools. These approaches all have achieved state-of-the-art performance.

Oyente (Luu et al. 2016), Mythril (ConsenSys 2019), Securify (Tsankov et al. 2018), and Pluto (Ma et al. 2022) are popular bytecode level static analysis tools that rely on fixed expert rules. They recover bytecode to a higher representation, which can comprehensively analyze all smart contracts in the blockchain. Oyente, Mythril, and Securify are common baselines and widely applied in static analysis (Liu et al. 2023; Ma et al. 2022; Perez and Livshits 2021). Pluto considers inter-contract scenarios and shows great outperformance in vulnerability detection.

TMP (Zhuang et al. 2020), AME (Liu et al. 2021), and CGE (Liu et al. 2023) are source code level analysis tools that explore the power of graph neural network models to detect vulnerabilities. Source code level tools can use rich information, including variable names, function names, and specific locations of smart contracts from the source code. With clear data flow and complete control flow, they easily achieve better performance.

Hyperparameter settings In Table 4, we list the hyperparameter settings for the deep learning models, i.e., CNN, GraphSAGE, and GAT, employed in the ByteEye. Furthermore, we provide reasonable ranges of hyperparameter settings and other details on our open-source repository. To achieve better performance, we also employ the Adam optimizer, the MultiStepLR/CosineAnnealingLR learning rate schedulers, and the CrossEntropyLoss/NLL-Loss loss functions in the models.

4.2 Effectiveness of individual modules

We perform a set of ablation experiments to evaluate what each individual module contributes to the effectiveness of ByteEye. We mainly focus on the edge-enhanced CFG construction module, the feature extraction and encoding module, and the vulnerability detection module of ByteEye. The results highlight that the three key parts can individually contribute to the effectiveness of ByteEye, and they achieve the best performance when combined together. For fair comparison and the pursuit of higher performance, we should evaluate the models of vulnerability detection first to select the best one as the starting point of subsequent experiments.

Results on machine learning models We first test the vulnerability detection models, as shown in Table 5. ByteEye(SVM), ByteEye(RF), ByteEye(KNN), ByteEye(CNN), ByteEye(GraphSAGE), and ByteEye(GAT) represent the configurations for SVM,

Table 4 Hyperparameter Settings for the Deep Learning Models

Models	<i>lr</i>	<i>p</i>	<i>bs</i>	<i>epoch</i>	<i>lr</i>	<i>p</i>	<i>bs</i>	<i>epoch</i>	<i>lr</i>	<i>p</i>	<i>bs</i>	<i>epoch</i>
	RE (SCP Dataset)				TD (SCP Dataset)				IO (SCP Dataset)			
CNN	0.005	0.2	16	200	0.002	0.1	16	200	0.002	0.1	16	200
GraphSAGE	0.025	0.2	16	250	0.02	0.01	16	200	0.02	0.2	8	300
GAT	0.0165	0.21	16	500	0.017	0.19	8	500	0.015	0.08	16	500
	RE (ESCP Dataset)				TD (ESCP Dataset)				IO (ESCP Dataset)			
CNN	0.02	0.2	16	200	0.0002	0.2	16	200	0.002	0.2	16	200
GraphSAGE	0.014	0.2	8	700	0.01	0.02	16	200	0.008	0.2	16	500
GAT	0.0285	0.011	16	500	0.035	0.1	16	500	0.024	0.012	16	500

The lr, p, and bs denote the learning rate, dropout rate, and batch size, respectively

Table 5 The Ablation Study Results on Machine Learning Models

Methods	<i>Acc.</i> (%)	<i>Prec.</i> (%)	<i>Rec.</i> (%)	<i>F1</i> (%)	<i>Acc.</i> (%)	<i>Prec.</i> (%)	<i>Rec.</i> (%)	<i>F1</i> (%)
RE (SCP Dataset)					RE (ESC_P Dataset)			
ByteEye(SVM)	92.61	93.48	53.75	68.25	80.59	95.45	28.77	44.21
ByteEye(RF)	91.50	74.29	65.00	69.33	82.05	76.09	47.95	58.82
ByteEye(KNN)	93.35	77.50	77.50	77.50	76.92	60.00	41.10	48.78
ByteEye(CNN)	92.24	73.17	75.00	74.07	82.05	66.22	67.12	66.67
ByteEye(GraphSAGE)	91.68	68.82	80.00	73.99	81.69	68.25	58.90	63.24
ByteEye(GAT)	97.41	94.59	87.50	90.91	93.41	88.73	86.30	87.50
TD (SCP Dataset)					TD (ESC_P Dataset)			
ByteEye(SVM)	79.86	54.17	21.67	30.95	70.45	65.00	50.98	57.14
ByteEye(RF)	81.25	58.82	33.33	42.55	68.94	65.63	41.18	50.60
ByteEye(KNN)	77.78	47.14	55.00	50.77	61.36	50.00	54.90	52.34
ByteEye(CNN)	78.82	49.06	43.33	46.02	69.70	60.78	60.78	60.78
ByteEye(GraphSAGE)	82.99	70.37	31.67	43.68	68.94	61.90	50.98	55.91
ByteEye(GAT)	82.29	56.34	66.67	61.07	75.00	69.57	62.75	65.98
IO (SCP Dataset)					IO (ESC_P Dataset)			
ByteEye(SVM)	86.71	100.00	16.00	27.59	71.64	58.82	12.35	20.41
ByteEye(RF)	86.08	66.67	24.00	35.29	74.18	64.71	27.16	38.26
ByteEye(KNN)	78.80	36.07	44.00	39.64	69.45	47.62	37.04	41.67
ByteEye(CNN)	83.54	47.73	42.00	44.68	75.64	61.29	46.91	53.15
ByteEye(GraphSAGE)	85.76	56.76	42.00	48.28	76.73	79.31	28.40	41.82
ByteEye(GAT)	88.61	67.50	54.00	60.00	84.36	78.79	64.20	70.75

RF, KNN, CNN, GraphSAGE, and GAT models, respectively. Making full use of global graph structure information, employing GNN models roughly outperforms employing other models in ByteEye.

ByteEye(GAT) achieves the overall best performance among all models since it considers the different importance of different nodes. For example, ByteEye using GAT achieves an average of 18.22% on F1 higher than using the GraphSAGE model. ByteEye(GAT) utilizes multi-attention mechanisms to assign different importance to nodes of the same neighborhood, thus it can pay more attention to the crucial information related to vulnerabilities. However, the GraphSAGE model only randomly aggregates neighbors' information and cannot remember important node information during iterations, thus failing to strengthen crucial information that contains vulnerability knowledge. While the accuracy can be improved by increasing the number of neighbor nodes, we need to consider the trade-off between accuracy and computational efficiency.

As to CNN, it also has poor performance compared with GAT. Because CNN fails to process the global graph structure information and has trouble in transferring nodes' crucial information to neighbors as the graph denoted, employing CNN will cause the loss of accuracy. Due to the above reasons, we select GAT as our ByteEye default detection model for subsequent experiments.

Results on edge-enhanced CFG and features Table 6 shows the results of the ablation experiment on edge-enhanced CFG and features. ByteEye-CFG represents the

configuration where we disable the partial symbolic execution phase in the CFG construction and only rely on pattern recognition. ByteEye-FM1, ByteEye-FM2, ByteEye-FM3, and ByteEye-FM4 represent removing the statistics information, spatial structure information, key instruction information, and protection mechanism information from ByteEye, respectively. To further analyze edge-enhanced CFG and features of ByteEye, as illustrated in Fig. 8, we visualize the partial ablation results, such as the precision and recall of Table 6, showing the improvement in the corresponding measurements. Taking Fig. 8 (a) as an example, on the SCP dataset, the improvement rate of precision on reentrancy is defined as the precision of ByteEye* on reentrancy minus precision of ByteEye-CFG on reentrancy.

As demonstrated in Table 6, the edge-enhanced CFG is effective in reducing both false negatives and false positives. We observe a significant increase in the recall and precision of edge-enhanced CFG (ByteEye*) for all vulnerabilities on both datasets, compared with ByteEye-CFG, which only uses pattern recognition to construct CFG. In particular, edge-enhanced CFG contributes more to reducing false negatives, since enhanced edges help maintain rich information from smart contracts. In Fig. 8(a), the addition of the edge-enhanced CFG part results in an average improvement rate of 11.53% on recall, more than twice the improvement rate on precision.

Each type of feature module individually contributes to the improvement of detection effectiveness, which is shown in Table 6 and Fig. 8(b)-(e). The statistics infor-

Table 6 The Ablation Study Results on edge-enhanced CFG and Features

Methods	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)	Acc. (%)	Prec. (%)	Rec. (%)	F1 (%)
RE (SCP Dataset)				RE (ESC _P Dataset)				
ByteEye-CFG	95.75	91.30	78.75	84.56	89.74	83.58	76.71	80.00
ByteEye-FM1	93.72	92.59	62.50	74.63	91.21	83.56	83.56	83.56
ByteEye-FM2	93.35	79.73	73.75	76.62	87.91	80.30	72.60	76.26
ByteEye-FM3	94.27	88.89	70.00	78.32	90.11	84.85	76.71	80.58
ByteEye-FM4	95.19	87.50	78.75	82.89	86.45	69.57	87.67	77.58
ByteEye-* ¹	97.41	94.59	87.50	90.91	93.41	88.73	86.30	87.50
TD (SCP Dataset)				TD (ESC _P Dataset)				
ByteEye-CFG	80.56	54.00	45.00	49.09	68.94	62.50	49.02	54.95
ByteEye-FM1	78.13	46.67	35.00	40.00	64.39	57.69	29.41	38.96
ByteEye-FM2	79.17	50.00	48.33	49.15	72.73	68.29	54.90	60.87
ByteEye-FM3	79.51	60.00	5.00	9.23	64.39	55.00	43.14	48.35
ByteEye-FM4	80.56	54.00	45.00	49.09	70.45	62.50	58.82	60.61
ByteEye-* ¹	82.29	56.34	66.67	61.07	75.00	69.57	62.75	65.98
IO (SCP Dataset)				IO (ESC _P Dataset)				
ByteEye-CFG	87.03	62.16	46.00	52.87	80.73	71.88	56.79	63.45
ByteEye-FM1	86.39	64.00	32.00	42.67	77.09	66.07	45.68	54.01
ByteEye-FM2	86.08	56.82	50.00	53.19	77.09	65.00	48.15	55.32
ByteEye-FM3	86.39	70.59	24.00	35.82	70.18	49.15	35.80	41.43
ByteEye-FM4	87.34	64.71	44.00	52.38	80.00	68.57	59.26	63.58
ByteEye-* ¹	88.61	67.50	54.00	60.00	84.36	78.79	64.20	70.75

¹* represents our method with three modules, including edge-enhanced CFG, features, and the GAT model

mation (FM1) and spatial structure (FM2) contribute to reducing both false positives and false negatives due to the general behaviors they capture. FM1 and FM2 achieve 22.21% and 12.28% improvement on recall, and 7.49% and 9.23% improvement on precision, respectively. The key instructions (FM3) that relate to specific vulnerability information contribute largely to reducing false negatives. In Fig. 8(d), FM3 achieves an average improvement of 27.80% on recall for all types of vulnerabilities. The protection mechanisms (FM4) show more contributions to reducing false positives than reducing false negatives. Especially for the reentrancy vulnerability on ESC_P dataset, disabling FM4 results in a decrease of 19.16% on precision, and an improvement of 1.37% on recall.

4.3 Effectiveness of the ByteEye framework

We compare ByteEye with seven baseline approaches, and experiment results highlight that ByteEye can consistently perform better on accuracy, F1, and success rate.

Table 7 shows the vulnerability detection results of all the compared baselines on both the ESC_P and the SCP dataset. Source code level analysis tools TMP, AME, and CGE use GNN models and are conducted by five-fold cross-validation. To eliminate the effect of data partition bias on small datasets, the average result of the five-fold is reported. We also prioritize maintaining the default configuration of hyperparameters in TMP, AME, and CGE. When hyperparameter tuning is required to accommodate dataset variations, we employ the criterion of attaining approximately 100% accuracy on the training set. For four bytecode level analysis tools, they rely on deterministic expert rules and are directly applied to the whole dataset.

Results on reentrancy ByteEye outperforms all baselines on both datasets in terms of F1 and *success rate*.

ByteEye employs well-formed edge-enhanced CFG and carefully designed features that consider both statistical and instruction features, and protection mechanisms. With these unique designs, ByteEye achieves the highest F1. The four approaches, Mythril, Oyente, Securify, and Pluto, which rely on expert rules, perform relatively low F1 in vulnerability detection. The reason is that the same type of vulnerabilities usually have different forms of presentation, making it hard to enumerate with expert rules. Although TMP, AME, and CGE with well-designed features are more generalizable to different vulnerability forms, they lack enough expert knowledge and protection mechanisms in design. Therefore, ByteEye performance is also better than that of the three approaches.

In terms of *success rate*, ByteEye is able to predict all smart contracts on both datasets. However, TMP, AME, and CGE with rich source code information show the lowest success rate on the SCP dataset. We find that they only identify the keyword `call.value` to build the graph. For smart contracts that do not contain that keyword, the three methods produce an empty graph which results in errors.

It is worth mentioning that Pluto shows the lowest detection results (Precision, Recall, and F1) on both datasets. We check the open source implementation of Pluto and find that when Pluto meets STOP opcode, it views this as the end of the program

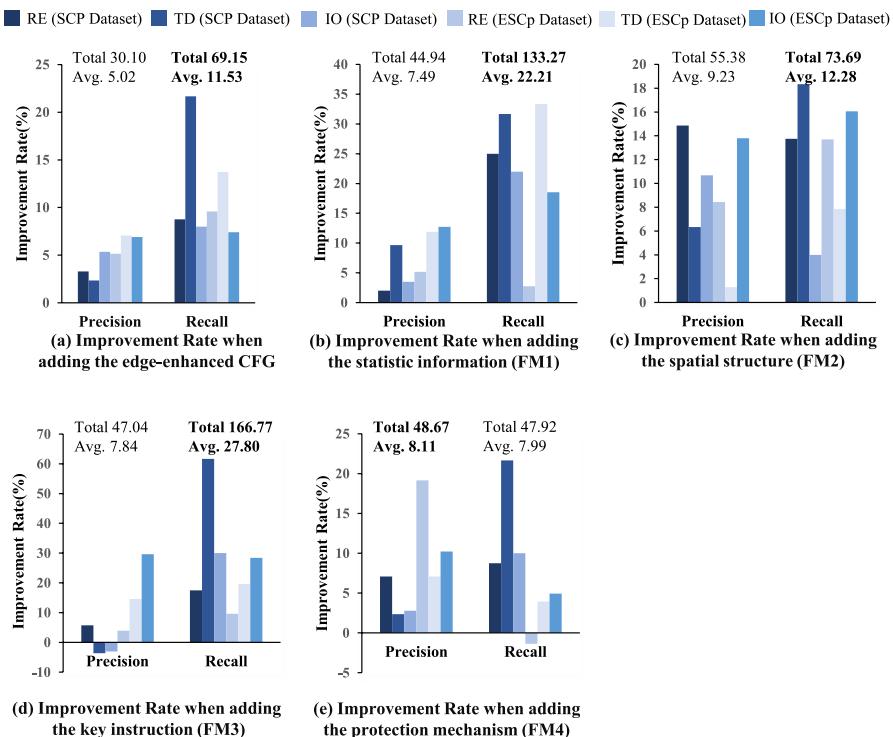


Fig. 8 The Visualization Result of the Ablation Results on edge-enhanced CFG and Features. (a) represents the improvement rate of precision and recall on two datasets and three vulnerabilities when adding the edge-enhanced CFG. (b), (c), (d), and (e) represent the improvement rate when employing the statistics information(FM1), the spatial structure(FM2), the key instruction(FM3), and the protection mechanism(FM4), respectively

and stops construction (confirmed with the authors of Pluto). Note that smart contracts may actually have multiple STOP opcodes, and thus Pluto only processes a small scale of bytecode and causes many false negatives.

Results on timestamp dependency ByteEye achieves the highest F1 on both datasets, proving its effectiveness in reducing false negatives and false positives. Specifically, for the ESC_P dataset, ByteEye achieves 65.98% F1, outperforming the second-place competitor CGE by 9.84%. For the SCP dataset, ByteEye achieves 61.07% F1, surpassing the second-place competitor Oyente by 2.08%. Notably, Securify can not detect timestamp dependency vulnerabilities.

For *success rate*, ByteEye also achieves 100%. Since ByteEye clearly splits the pure code segment and invalid data segments according to different versions of smart contracts, thus can analyze all smart contracts successfully. TMP, AME, and CGE perform the worst because they only identify the `block.timestamp` keyword when building the CFG, and overlook the other potential related keywords.

Table 7 The Vulnerability Detection Results

Methods	Acc. (%)		Prec. (%)		Rec. (%)		F1 (%)		Suc. (%)		Acc. (%)		Prec. (%)		Rec. (%)		F1 (%)		Succ. (%)		
	RE (SCP Dataset)		RE (ESC _P Dataset)		RE (SCP Dataset)		RE (ESC _P Dataset)		RE (ESC _P Dataset)		TD (SCP Dataset)		TD (ESC _P Dataset)		TD (SCP Dataset)		TD (ESC _P Dataset)		IO (SCP Dataset)		IO (ESC _P Dataset)
TMP	86.55	87.14	89.71	88.41	22.00	90.48	87.30	75.44	80.88	100.00	-	-	-	-	-	-	-	-	-	-	-
AME	85.71	85.50	89.39	87.41	22.00	91.94	93.22	75.34	83.33	100.00	-	-	-	-	-	-	-	-	-	-	-
CGE	88.24	88.57	91.18	89.86	22.00	92.31	86.11	84.93	85.52	100.00	-	-	-	-	-	-	-	-	-	-	-
Mythril	86.87	55.00	68.75	61.11	98.50	46.32	31.84	87.67	46.72	99.63	-	-	-	-	-	-	-	-	-	-	-
Oyente	85.21	50.00	71.25	58.76	100.00	65.57	42.11	76.71	54.37	100.00	-	-	-	-	-	-	-	-	-	-	-
Security	86.17	52.59	76.25	62.24	98.89	34.43	27.16	86.30	41.31	100.00	-	-	-	-	-	-	-	-	-	-	-
Pluto	80.78	7.14	2.53	3.74	99.08	49.45	9.88	10.96	10.39	100.00	-	-	-	-	-	-	-	-	-	-	-
ByteEye	97.41	94.59	87.50	90.91	100.00	93.41	88.73	86.30	87.50	100.00	-	-	-	-	-	-	-	-	-	-	-
TMP	87.34	54.55	19.35	28.57	82.29	64.77	48.65	60.00	53.73	66.67	-	-	-	-	-	-	-	-	-	-	-
AME	85.65	36.36	12.90	19.05	82.29	68.18	53.13	56.67	54.84	66.67	-	-	-	-	-	-	-	-	-	-	-
CGE	87.34	55.56	16.13	25.00	82.29	71.59	59.26	53.33	56.14	66.67	-	-	-	-	-	-	-	-	-	-	-
Mythril	75.75	36.84	11.67	17.72	93.06	66.15	100.00	12.00	21.43	98.49	-	-	-	-	-	-	-	-	-	-	-
Oyente	80.21	51.90	68.33	58.99	100.00	61.36	50.00	7.84	13.56	100.00	-	-	-	-	-	-	-	-	-	-	-
Security	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Pluto	79.58	53.85	23.33	32.56	98.61	67.24	66.66	10.00	17.39	87.88	-	-	-	-	-	-	-	-	-	-	-
ByteEye	82.29	56.34	66.67	61.07	100.00	75.00	69.57	62.75	65.98	100.00	-	-	-	-	-	-	-	-	-	-	-
TMP	-	-	-	-	-	82.45	70.37	70.37	70.37	100.00	-	-	-	-	-	-	-	-	-	-	-
AME	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
CGE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Mythril	78.26	40.51	64.00	49.61	94.62	82.18	71.62	65.43	68.39	100.00	-	-	-	-	-	-	-	-	-	-	-
Oyente	58.86	27.01	94.00	41.96	100.00	62.55	29.63	19.75	23.70	100.00	-	-	-	-	-	-	-	-	-	-	-
Security	74.83	28.81	34.00	31.19	94.30	49.45	36.45	96.30	52.88	100.00	-	-	-	-	-	-	-	-	-	-	-
Pluto	69.48	53.85	28.57	37.33	48.73	70.91	50.82	38.27	43.66	100.00	-	-	-	-	-	-	-	-	-	-	-
ByteEye	88.61	67.50	54.00	60.00	100.00	84.36	78.79	64.20	70.75	100.00	-	-	-	-	-	-	-	-	-	-	-

Results on integer overflow/underflow ByteEye achieves the best accuracy, precision, F1, and *success rate* on both datasets. Although Oyente achieves the highest recall on the SCP dataset and Security achieves the highest recall on the ESC_P dataset, their corresponding precisions are so low that they produce a large number of false positives, leading to massive human labor time to check. Because TMP pre-processed data on the ESC_P dataset is available, we can only report the execution results of TMP. The pre-processing procedures for extracting expert patterns of integer overflow/underflow are not described in TMP, AME, and CGE, so we cannot report results on the SCP dataset.

4.4 Real world vulnerability detection

To prove ByteEye can be applied in real-world scenarios, we perform experiments on the smart contracts collected from Ethereum, and report the vulnerabilities detected. In these real-world smart contracts, ByteEye can detect 10, 52, and 299 cases of unreported vulnerabilities including reentrancy, timestamp dependency, and integer overflow/underflow, respectively.

We collect the newest 33,120 smart contracts, and the related block numbers of which range from 15930K to 18081K. The latest compilation version of smart contracts is up to 0.8.x. Compared to the other types, the number of reentrancy is small in the newly released smart contracts, since developers widely adopt protection mechanisms. In contrast, most existing approaches do not make full use of these protection mechanisms, resulting in false positives.

Figure 9 shows a typical case study of a smart contract with the timestamp dependency vulnerability detected by ByteEye. The function `liquidate` calls function `noTransferRepayAtMaturity`. The actual vulnerability exists in line 10, where the timestamp can be manipulated to affect the value `actualRepayAssets`. The value of `actualRepayAssets` is the return value for the function `liquidate` (line 4) and affects the value of `repaidAssets` (line 4). Finally, the `repaidAssets` decides the transferred amount (line 7). This vulnerability cannot be detected by existing baseline approaches.

4.5 Discussion

To comprehensively explore and evaluate the performance of ByteEye, we further discuss both latency and the accuracy of CFG construction.

Latency It is important to analyze large-scale smart contracts with low latency, so we evaluate the latency of ByteEye in three aspects: overall runtime analysis, CFG construction latency, and a scalability test of inference latency under different batches of smart contracts. For runtime analysis, we compare ByteEye with the representative symbolic execution baseline Pluto. We choose Pluto as our comparison representative because it achieves low latency in a large-scale latency analysis of 39,443 smart contracts. Specifically, Pluto costs an average of 16.9 seconds to analyze per smart contract, whereas Mythril and Security cost 231.8 seconds and 40.4 seconds, respectively (Ma et al. 2022). Table 8 shows the runtime analysis of ByteEye compared with

```

01 contract Market{
02     function liquidate(...) external returns (uint256 repaidAssets) {
03         while (...) {
04             actualRepay = noTransferRepayAtMaturity(...);
05             repaidAssets += actualRepay;
06         }
07         asset.safeTransferFrom(msg.sender, ..., repaidAssets + lendersAssets);
08     }
09     function noTransferRepayAtMaturity(...) internal
10        returns (uint256 actualRepayAssets ) {
11         // early repayment allows a discount from the unassigned earnings
12         if(block.timestamp < maturity) {
13             .....
14             actualRepayAssets = debtCovered - discountFee;
15         } else {
16             actualRepayAssets = debtCovered + debtCovered.mulWadDown\(
17                         ((block.timestamp - maturity) * penaltyRate);
18     } } }
```

Fig. 9 The Real Case of a Timestamp Dependency Vulnerability. This vulnerability is detected in market Smart Contract. The corresponding Ethereum address is 0xB49212AF04e9254678BE7EA3C754c1858977CFfe

Table 8 Runtime Analysis of ByteEye Compared with Pluto

	Number of smart contracts	Overall runtime	Average runtime	Average runtime
			without training stage	
Pluto	536	4184.75s	7.81s	7.81s
ByteEye	541	2691.48s	4.98s	0.12s
Speedups	-	1.55×	1.57×	65.08×

Pluto for the reentrancy vulnerability detection on the SCP dataset. The experiment results indicate that compared with Pluto, ByteEye can achieve $1.57\times$ and $65.08\times$ speedups on average analysis time and average analysis time without training stage, respectively. With a trained model, ByteEye only costs 0.12 seconds to analyze a smart contract, whereas Pluto costs 7.81 seconds to analyze a smart contract.

In Fig. 10, we further display the CFG construction latency speedups of ByteEye, compared with Pluto. Among them, Avg time / Path explosion denotes the average CFG construction time per contract without path explosion problems, and Avg time /low code coverage denotes the average CFG construction time per contract without low code coverage rate and path explosion problems. ByteEye can successfully analyze 541 smart contracts, whereas Pluto analyzes the same scale contracts, but five contracts face path explosion problems. Due to the premature stop and incomplete code pattern, Pluto performs low code coverage (lower than 30% as provided by Pluto) in 233 smart contracts. The experiment result also shows that ByteEye

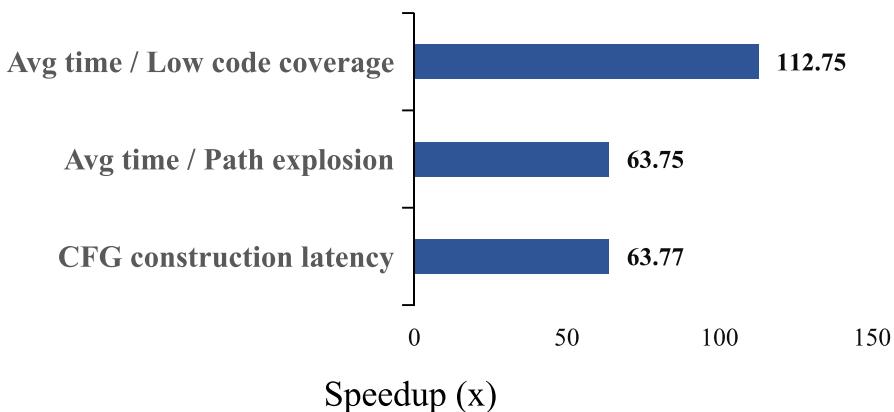


Fig. 10 The CFG Construction Latency Speedups of ByteEye Compared with Pluto

achieves $63.75\times$ and $112.75\times$ speedups on average time per contract without path explosion and average time per contract without low code coverage, respectively.

To evaluate the inference latency in large-scale smart contracts, we test the inference latency when detecting 1, 16, 160, 1,600, and 16,000 smart contracts, and repeat each group 5 times. Figure 11 shows average inference latency per contract under different batches of smart contracts. ByteEye costs 0.0682 seconds on average inference latency to analyze a smart contract, while it takes less average inference latency to analyze larger scales of smart contracts. When analyzing 16,000 smart contracts, ByteEye only costs 96.78406 seconds of inference latency in total.

Consequently, ByteEye achieves low latency compared with the representative symbolic execution method, and it costs 0.12 seconds to analyze a smart contract in

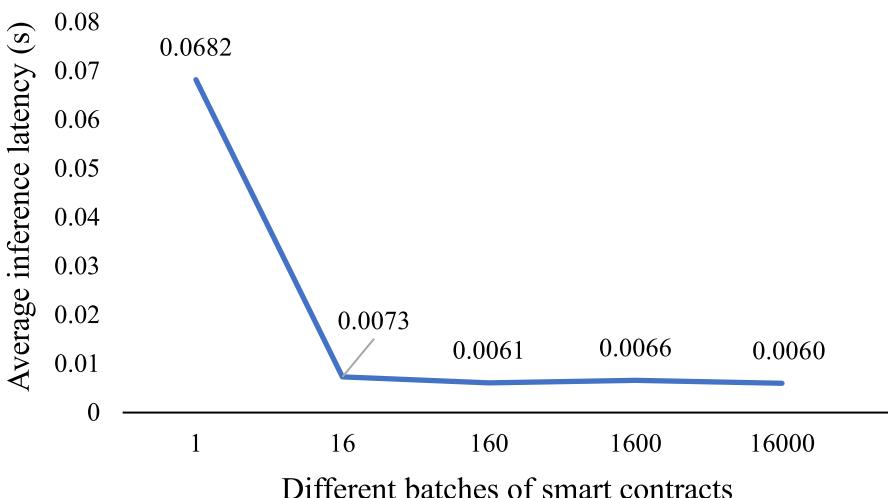


Fig. 11 Average Inference Latency per Contract under Different Batches of Smart Contracts

runtime analysis. ByteEye also shows a great advantage in CFG construction latency and the potential to detect large-scale smart contracts.

CFG construction accuracy To prove the quality of edge-enhanced CFG, we evaluate the CFG construction accuracy on dataset SolidiFI (Ghaleb and Pattabiraman 2020). SolidiFI dataset consists of 50 buggy contracts injected by 9369 bugs from seven different bug types, especially containing 1343 injected reentrancy bugs, which is suitable for a fast and comprehensive detection task.

Disabling the other two main modules, we detect vulnerabilities only based on edge-enhanced CFG and a concise CALL-SLOAD/SSTORE expert rule. We also keep the same settings with the SOTA CFG construction bytecode level method, EtherSolve (Contro et al. 2021). Followed by the reentrancy analysis experiment that EtherSolve conducted, we compare ByteEye with the aforementioned three methods and three new methods. In Fig. 12, Avg. detection per contract denotes the average number of detected bugs per contract, and Diff. with SolidiFI dataset denotes the average number of differences with the standard. The result shows that ByteEye achieves a -0.62 difference with standard SolidiFI injection. ByteEye can outperform the SOTA bytecode level method EtherSolve, because ByteEye builds all possible control flow jumps in the static analysis. Moreover, ByteEye shows the best performance in all compared bytecode level methods and even the second-best performance in all methods, including source code level methods.

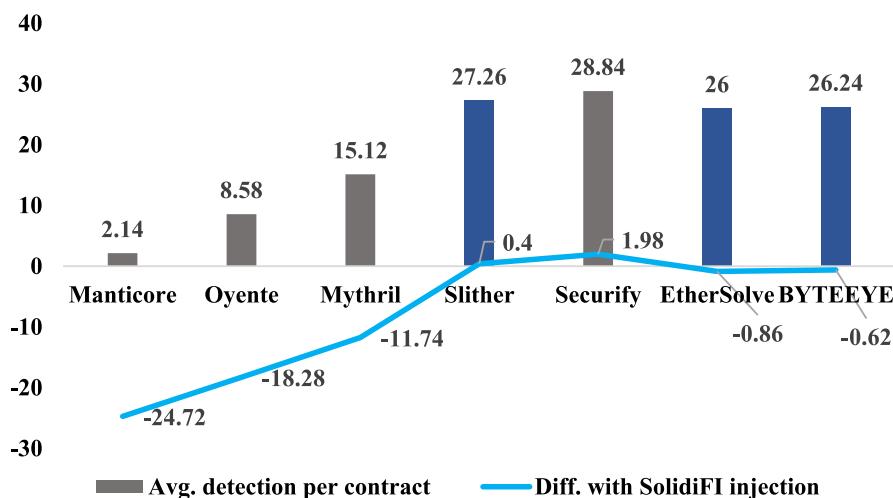


Fig. 12 Reentrancy Analysis Comparison on SolidiFI Dataset

5 Limitations

In this section, we discuss ByteEye’s limitations on generalization when detecting vulnerabilities beyond the evaluated types, robustness when facing bytecode modification or obfuscation, and interpretability of black-box models’ prediction results.

Generalizability The generalizability of ByteEye in detecting new vulnerability types is limited by (1) the accurate description of feature modules and (2) the scale of the specialized training dataset. As mentioned in Section 3.3, ByteEye can detect a new vulnerability class by updating the key instructions module and the protection mechanisms module. To further precisely detect, we need to analyze the Solidity code of the new type and then extract the suitable key instructions and the protection mechanism in the bytecode. The performance of ByteEye will largely benefit from the accurate description of the two modules related to the specific vulnerability type. Besides, ByteEye requires a specialized training dataset to learn the features of the new vulnerability type. Our practical experience proves that the training dataset needs to be large enough that have at least 50 positives. The performance of ByteEye will be improved by a large-scale specialized training set.

Robust For robustness of our feature design, we discuss two scenarios concerning facing adversarial bytecode modifications or obfuscations. If the smart contract remains executable after modification or obfuscation, our feature design is robust, capable of correctly identifying bytecode features. Conversely, if it becomes non-executable, the modified smart contract will not be deployed on the blockchain or analyzed by the vulnerability detection methods. Specifically, the two situations are: (1) the adversary changes the content of the source code and replaces the original bytecode with the changed compiled smart contract. In this situation, the feature design is robust enough to handle the changes of function names, variable names, and crucial call statements. Because the feature design at the bytecode level does not rely on the specific function names, variable names, or crucial call statements. For example, when the adversary changes the crucial variable names, the feature design still identifies the variable by the low-level opcode ‘CALLDATALOAD’, not the variable name ‘x’; (2) The adversary makes arbitrary modifications or obfuscations to the bytecode of the smart contract, like inserting a wrong hexadecimal string. In this situation, ByteEye is sufficiently robust that it will not build the CFG based on the modified bytecode, which is before the feature design stage. Since each opcode has a one-to-one correspondence with a two-digit hexadecimal value, a wrong and meaningless hexadecimal string cannot be identified by ByteEye at the building CFG stage.

Interpretability Given that GNNs can be black-box models, we discuss the interpretability of model predictions in three aspects: (1) GNNs has great potential in smart contract vulnerability detection, since their excellent feature extraction performance has been proven in various fields. GNNs is active in various fields such as node classification, link prediction, and anomaly detection; (2) The predictive performance of GNN models has been experimentally verified. At the same dataset and the same

measure metrics, the average detection performance of methods exploiting GNN models is better than traditional static analysis methods. (3) The visualization analysis result can prove that GNN models learn the actual and important information in the smart contract vulnerability detection task.

To explain the third aspect, we visualize the GAT model's attention weights from a simple smart contract and present Fig. 13 for illustration. Figure 13 mainly shows that the attention mechanism of the GAT model helps us to pay more attention to some nodes that contain important information. In Fig. 13(a), we have presented a heatmap that annotates the trained attention weights of each node (i.e., block) in a CFG. The crucial area in the heatmap is marked by the red box, and the range of which comes from node 13 to node 17. In the red box, node 13 puts 0.97 of attention weight on node 15, meaning that ByteEye pays more attention to the route of node 13 to node 15, rather than the route of node 13 to node 14. Node 15 puts 0.48 of attention weight on itself, and 0.50 of attention weight on node 16, meaning that as to node 15, the information of node 15 and node 16 is much more important than node 17. We further denote the partial CFG containing node 13-node 17 in Fig. 13(b). Strengthened by the attention weights shown in Fig. 13(a), we pay close attention to the route from node 13 to node 15, and node 15 to node 16. Note that node 13 and node 16 contain important opcode instructions related to the specific vulnerability. Therefore, ByteEye correctly identifies the important nodes with the help of the attention mechanism.

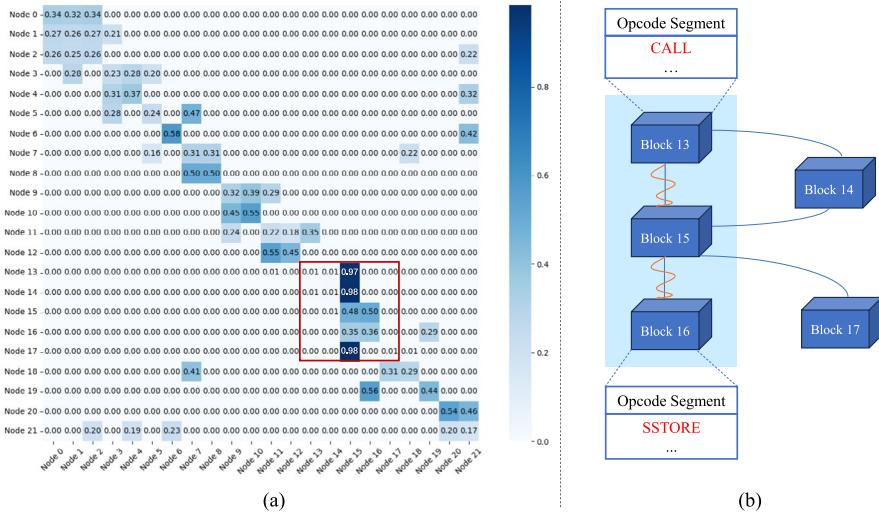


Fig. 13 The Visualization of GAT's Attention Weights

6 Related work

Bytecode level detection methods There are approaches that usually combine static program analysis techniques with fixed expert rules for vulnerability detection, such as symbolic execution, taint analysis, and formal method. Oyente (Luu et al. 2016) is designed with operation semantics and symbolic execution. Other symbolic execution methods that followed include teEther (Krupp and Rossow 2018), EthIR (Albert et al. 2018), and SAILFISH (Bose et al. 2022). Besides, Vandal (Brent et al. 2018), Securify (Tsankov et al. 2018), and eThor (Schneidewind et al. 2020) utilize logic-based analysis or the formal method to define accurate expert rules to detection. In taint analysis, Ethainter (Brent et al. 2020) checks information flow with data sanitization in smart contracts. SmartDagger (Liao et al. 2022) recovers bytecode semantics and identifies vulnerabilities based on taint propagation. There are also approaches that combine static analysis techniques with dynamic information or on-chain status for better performance. TXSPECTOR (Zhang et al. 2020) performs well by exploiting transaction traces with EVM dynamic execution information. Pluto (Ma et al. 2022) employs on-chain information to detect new vulnerabilities. These approaches that rely on fixed complex expert rules usually suffer from extensibility issues. Manually crafted expert rules make it hard to process new forms of vulnerabilities. In addition, there are also approaches utilizing dynamic symbolic execution or fuzzing to test or analyze smart contracts. Sereum (Rodler et al. 2019) and SODA (Chen et al. 2020) focus on runtime vulnerability detection. Contractfuzzer (Jiang et al. 2018), sFuzz (Nguyen et al. 2020), and ETHBMC (Frank et al. 2020) detect vulnerabilities precisely with runtime information during dynamic execution. However, these execution approaches are time-consuming. The latest approaches employ machine learning models to analyze bytecode. ContractWard (Wang et al. 2020) tries to solve the extensibility issue of rule-based approaches, but it simply splits the bytecode into sliced windows as features that lack vulnerability-specific knowledge. Vulhunter (Li et al. 2023) utilizes multi-instance learning to reason instance labels and capture features, while it does not incorporate comprehensive expert knowledge such as protection mechanisms, which ByteEye employs to reduce false positives.

Source code level detection methods Static analysis tools such as Smartcheck (Tikhomirov et al. 2018), Slither (Feist et al. 2019), ZEUS (Kalra et al. 2018), and Clairvoyance (Xue et al. 2020) use pattern matching or symbolic analysis and rely on fixed expert rules for vulnerability detection. Slither conducts a thorough analysis of the Static Single Assignment (SSA) format for vulnerability detection. Dynamic analysis tools such as Confuzzius (Torres et al. 2021) and Echidna-parade (Groce and Grieco 2021) utilize the fuzzing technique to detect vulnerabilities, but they are limited by the large execution latency overhead. Some approaches also exploit ML models to detect vulnerabilities. They extract semantic information from source code to construct the graph, and design different GNN models to detect vulnerabilities. Zhuang et al. (2020) propose to use the graph neural network in smart contract vulnerability detection for higher accuracy and efficiency. They convert the smart contract source code into small graphs and then use the GCN model with no degree (DR-GCN) or with temporal information for vulnerability detection (TMP). Their recent works,

AME (Liu et al. 2021) and CGE (Liu et al. 2023), further combine TMP and expert rules to achieve better accuracy. Cai et al. (2023) present an approach to enhancing TMP in graph representation and the program slicing technique. SCVHunter (Luo et al. 2024) converts source code into intermediate representations to construct a semantic graph and utilize the heterogeneous graph attention network to detect vulnerabilities.

As a bytecode level detection method, ByteEye utilizes bytecode as input and can enable most smart contracts detection on Ethereum without source code. Moreover, through incorporating both control flow information and carefully designed features, ByteEye can achieve superior performance even compared to SOTA source code level methods.

7 Threats to validity

The *internal threat to validity* mainly lies in the implementations of ByteEye and the labelling of datasets. We have conducted decent testing as well as peer code review to ensure the quality of our implementation. Regarding the dataset labelling, we follow the standard labelling process, where two experts labelled the same smart contract independently. Then the labels are checked for consensus. In case of any inconsistent labelling, discussions are conducted to achieve consensus.

The *external threat to validity* mainly lies in the tools that we compared. To eliminate the threats, we carefully check the manual of the released tools and try to repeat the results reported in their papers. In case of inconsistent results, e.g., the low success rate for TMP, AME, and CGE on the SCP dataset and the poor F1 of Pluto, we manually check for the reasons and contact the authors to confirm the issues.

The *construct threats to validity* may be caused by randomness during the training of the machine learning models. To eliminate this threat, we conduct the 5-fold cross-validation and report the average experiment results. We evaluate ByteEye with two datasets due to the limited availability of open source datasets. To further evaluate the effectiveness and the generalizability of ByteEye, we conduct an extra experiment that uses ByteEye to detect vulnerabilities in real world.

8 Conclusion and future works

In this paper, we present an extensible bytecode level smart contract vulnerability detection framework. The framework named ByteEye contains three key parts. The edge-enhanced CFG is effective in reducing false negatives and achieves an average increment of 11.53% on recall for all types of vulnerabilities. The three feature extraction modules contribute to an average improvement of 22.21%, 12.28%, and 27.80% in recall for all types of vulnerabilities, respectively. The protection mechanisms feature module achieves an average improvement of 8.11% in precision. Our ByteEye integrated with GAT can outperform seven comparison approaches. Especially, ByteEye performs an average of 35.29%, 43.95%, and 6.38% higher on the F1

score than the bytecode level best-performed baseline on three typical types of vulnerability detection, respectively. In future studies, we intend to expand our dataset by collecting a greater number of smart contracts, thereby enhancing the diversity of our data. This will facilitate the identification of new vulnerabilities. Additionally, we plan to design graph neural network models tailored for smart contract vulnerability detection.

Acknowledgements This work is partially supported by the National Natural Science Foundation (NO.62372253), the Natural Science Foundation of Tianjin Fund (23JCYBJC00010), and the Open Fund of Civil Aviation Smart Airport Theory and System Key Laboratory, Civil Aviation University of China (NO.SATS202303).

Author Contributions Jinni Yang participated in conceptualization, methodology design, data collection, experiments, and manuscript writing. Liu Shuang contributed to conceptualization, methodology designs, and manuscript writing and supervised the entire project. Surong Dai assisted in data collection and procession and contributed to the proofreading of the manuscript. Yaozheng Fang and Kunpeng Xie contributed to the revision and proofreading of the manuscript. Ye Lu secured funding for the research, provided resources, assisted in editing the manuscript, and supervised the entire project. All authors reviewed and approved the final version of the manuscript.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>.

References

- Albert, E., Gordillo, P., et al.: Ethir: A framework for high-level analysis of Ethereum bytecode. In: Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings, pp. 513–520 (2018). Springer
- Bose, P., Das, D., Chen, Y., et al.: Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022, pp. 161–178 (2022)
- Breiman, L.: Random forests. *Mach. Learn.* 5–32 (2001)
- Brent, L., Grech, N., Lagouvardos, S., et al.: Ethainter: A smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, pp. 454–469 (2020)

- Brent, L., Jurisevic, A., et al.: Vandal: A scalable security analysis framework for smart contracts. [arXiv:1809.03981](https://arxiv.org/abs/1809.03981) (2018)
- Cai, J., Li, B., Zhang, J., et al.: Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *J. Syst. Softw.* **195**, 111550 (2023)
- Chen, T., Cao, R., Li, T., et al.: Soda: A generic online detection framework for smart contracts. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020 (2020)
- Chen, T., Li, Z., et al.: A large-scale empirical study on control flow identification of smart contracts. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19-20, 2019, pp. 1–11 (2019)
- ConsenSys: Mythril. <https://github.com/ConsenSys/mythril-classic> (2019)
- Contro, F., et al.: Ethersolve: Computing an accurate control-flow graph from Ethereum bytecode. In: 29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021, pp. 127–137 (2021)
- Cover, T., Hart, P.: Nearest neighbor pattern classification. *IEEE Trans. Inf. Theory* **13**(1), 21–27 (1967)
- Durieux, T., Ferreira, J.F., Abreu, R., et al.: Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pp. 530–541 (2020)
- Feist, J., Grieco, G., Groce, A.: Slither: A static analysis framework for smart contracts. In: Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019, pp. 8–15 (2019)
- Feng, Q., Zhou, R., Xu, C., et al.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016, pp. 480–491 (2016)
- Frank, J., Aschermann, C., Holz, T.: Ethbmc: A bounded model checker for smart contracts. In: 29th USENIX Security Symposium, USENIX Security 2020, August 12–14, 2020, pp. 2757–2774 (2020)
- Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: ISSTA ’20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18–22, 2020, pp. 415–427 (2020)
- Grech, N., Brent, L., et al.: Gigahorse: Thorough, declarative decompilation of smart contracts. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019, pp. 1176–1186 (2019)
- Grech, N., Kong, M., Jurisevic, A., et al.: Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27 (2018)
- Groce, A., Grieco, G.: echidna-parade: A tool for diverse multicore smart contract fuzzing. In: ISSTA ’21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11–17, 2021, pp. 658–661 (2021)
- Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA (2017)
- Huang, J., Han, S., You, W., et al.: Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Trans. Inf. Forensics Secur.* **16**, 2144–2156 (2021). <https://doi.org/10.1109/TIFS.2021.3050051>
- Jiang, B., et al.: Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018, pp. 259–269 (2018)
- Kalra, S., Goel, S., Dhawan, M., et al.: Zeus: Analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18–21, 2018, pp. 1–12 (2018)
- Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017, Conference Track Proceedings (2017)
- Krupp, J., Rossow, C.: teether: Gnawing at Ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018, pp. 1317–1333 (2018)
- Lecun, Y., Bottou, L., Bengio, Y., et al.: Gradient-based learning applied to document recognition. *Proc. IEEE* **86**(11), 2278–2324 (1998). <https://doi.org/10.1109/5.726791>

- Li, Z., Lu, S., Zhang, R., et al.: Smartfast: An accurate and robust formal analysis tool for Ethereum smart contracts. *Empir. Softw. Eng.* **27**(7), 197 (2022)
- Li, Z., Lu, S., Zhang, R., et al.: Vulnhunter: Hunting vulnerable smart contracts at evm bytecode-level via multiple instance learning. *IEEE Trans. Software Eng.* **49**(11), 4886–4916 (2023). <https://doi.org/10.1109/TSE.2023.3317209>
- Liao, Z., Zheng, Z., Chen, X., et al.: Smartdagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability. In: ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022, pp. 752–764 (2022)
- Liu, Z., Qian, P., Wang, X., et al.: Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19–27 August 2021, pp. 2751–2759 (2021)
- Liu, Z., Qian, P., Wang, X., et al.: Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Trans. Knowl. Data Eng.* **35**(2), 1296–1310 (2023)
- Luo, F., Luo, R., Chen, T., Qiao, A., He, Z., Song, S., Jiang, Y., Li, S.: Sevhunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024, pp. 170–117013. ACM (2024). <https://doi.org/10.1145/3597503.3639213>
- Luo, X., Zhao, Y., Qin, Y., et al.: Towards semi-supervised universal graph classification. *IEEE Trans. Knowl. Data Eng.* **36**(1), 416–428 (2024). <https://doi.org/10.1109/TKDE.2023.3280859>
- Luu, L., Chu, D.-H., Olickel, H., et al.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC CCS, pp. 254–269 (2016)
- Ma, F., Xu, Z., Ren, M., et al.: Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Trans. Softw. Eng.* **48**(11), 4380–4396 (2022). <https://doi.org/10.1109/TSE.2021.3117966>
- Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
- Nguyen, T.D., Pham, L.H., et al.: sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In: ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, pp. 778–788 (2020)
- Perez, D., Livshits, B.: Smart contract vulnerabilities: Vulnerable does not imply exploited. In: 30th USENIX Security Symposium, USENIX Security 2021, August 11–13, 2021, pp. 1325–1341 (2021)
- Rodler, M., Li, W., Karame, G.O., et al.: Sereum: Protecting existing smart contracts against re-entrancy attacks. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24–27, 2019 (2019)
- Schneidewind, C., Grishchenko, I., Scherer, M., et al.: ethor: Practical and provably sound static analysis of Ethereum smart contracts. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9–13, 2020, pp. 621–640 (2020)
- Siegel, D.: Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists/> (2018)
- Solidity programming language. <https://soliditylang.org/> (2023)
- Suykens, J.A., Vandewalle, J.: Least squares support vector machine classifiers. *Neural Process. Lett.* **9**, 293–300 (1999)
- Tikhomirov, S., et al.: Smartcheck: Static analysis of Ethereum smart contracts. In: 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018, pp. 9–16 (2018)
- Torres, C.F., et al.: Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In: IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6–10, 2021, pp. 103–119 (2021)
- Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03–07, 2018, pp. 664–676 (2018)
- Tsankov, P., Dan, A., Drachsler-Cohen, D., et al.: Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018, pp. 67–82 (2018)
- Velickovic, P., Cucurull, G., Casanova, A., et al.: Graph attention networks. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings (2018)
- Wang, W., Song, J., Xu, G., et al.: Contractward: Automated vulnerability detection models for Ethereum smart contracts. *IEEE Trans. Netw. Sci. Eng.* **8**(2), 1133–1144 (2020)

- Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* **151**, 1–32 (2014)
- Xue, Y., Ma, M., Lin, Y., et al.: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21–25, 2020, pp. 1029–1040 (2020). IEEE
- Yaga, D., Mell, P., Roby, N., et al.: Blockchain technology overview. National Institute of Standards and Technology 8202 (2019)
- Zhang, M., Zhang, X., et al.: Txspector: Uncovering attacks in Ethereum from transactions. In: 29th USENIX Security Symposium, USENIX Security 2020, August 12–14, 2020, pp. 2775–2792 (2020)
- Zhuang, Y., Liu, Z., Qian, P., et al.: Smart contract vulnerability detection using graph neural network. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, pp. 3283–3290 (2020)
- Zou, W., Lo, D., et al.: Smart contract development: Challenges and opportunities. *IEEE Trans. Softw. Eng.* **47**(10), 2084–2106 (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Jinni Yang¹ · Shuang Liu³ · Surong Dai¹ · Yaozheng Fang¹ · Kunpeng Xie¹ · Ye Lu²

- ✉ Shuang Liu
shuang.liu@ruc.edu.cn
- ✉ Ye Lu
luye@nankai.edu.cn

¹ College of Computer Science, Nankai University, Tianjin, China

² College of Cryptology and Cyber Science, Nankai University, Tianjin, China

³ School of Information, Renmin University of China, Beijing, China