future internet

MDPI

Article

Toward Vulnerability Detection for Ethereum Smart Contracts Using Graph-Matching Network

Yujian Zhang [1,2,*] and Daifu Liu [1]

[1] School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China
[2] Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, Nanjing 211189, China
* Correspondence: yjzhang@seu.edu.cn

Abstract: With the blooming of blockchain-based smart contracts in decentralized applications, the security problem of smart contracts has become a critical issue, as vulnerable contracts have resulted in severe financial losses. Existing research works have explored vulnerability detection methods based on fuzzing, symbolic execution, formal verification, and static analysis. In this paper, we propose two static analysis approaches called *ASGVulDetector* and *BASGVulDetector* for detecting vulnerabilities in Ethereum smart contacts from source-code and bytecode perspectives, respectively. First, we design a novel intermediate representation called abstract semantic graph (ASG) to capture both syntactic and semantic features from the program. ASG is based on syntax information but enriched by code structures, such as control flow and data flow. Then, we apply two different training models, i.e., graph neural network (GNN) and graph matching network (GMN), to learn the embedding of ASG and measure the similarity of the contract pairs. In this way, vulnerable smart contracts can be identified by calculating the similarity to labeled ones. We conduct extensive experiments to evaluate the superiority of our approaches to state-of-the-art competitors. Specifically, *ASGVulDetector* improves the best of three source-code-only static analysis tools (i.e., *SmartCheck*, *Slither*, and *DR-GCN*) regarding the F1 score by 12.6% on average, while *BASGVulDetector* improves that of the three detection tools supporting bytecode (i.e., *ContractFuzzer*, *Oyente*, and *Securify*) regarding the F1 score by 25.6% on average. We also investigate the effectiveness and advantages of the GMN model for detecting vulnerabilities in smart contracts.

Keywords: smart contract; vulnerability detection; static analysis; abstract semantic graph; graph-matching network

Citation: Zhang, Y.; Liu, D. Toward Vulnerability Detection for Ethereum Smart Contracts Using Graph-Matching Network. *Future Internet* **2022**, *14*, 326. https://doi.org/10.3390/fi14110326

Academic Editors: Peter Kieseberg and Thomas Moser

Received: 6 October 2022
Accepted: 9 November 2022
Published: 11 November 2022

1. Introduction

Powered by the emerging technique of blockchain (e.g., Ethereum [1]), smart contracts, the concept of which was first raised in the late 1990s [2], have been reactivated and applied in various commercial fields, such as financial trades, supply chains, e-voting, etc. Technically, smart contracts are executable codes that run on top of the blockchain to facilitate, execute, and enforce an agreement between untrustworthy parties [3]. These codes and terms of the agreement therein are recorded in a distributed and public ledger, inheriting the immutable and distributed natures of blockchain [4]. After being deployed, smart contracts are triggered by transaction events, and enable decentralized applications running on a virtual executing environment (i.e., sandboxed environment) provided by the blockchain. For example, Ethereum [1] provides an ecosystem for smart contracts to develop decentralized applications using solidity and the Ethereum virtual machine (EVM). The former is a specific programming language to write smart contracts, while the latter is a Turing-complete virtual machine integrated into the Ethereum to execute contract bytecodes.

Like other computer programs, smart contracts also suffer from security vulnerabilities. In fact, many security incidents have happened to smart contracts, resulting in enormous

financial losses. In 2016, the notorious DAO attack due to the reentrancy vulnerability in Ethereum caused losses of ETC 3.6 million (i.e., cryptocurrencies of Ethereum) worth over CNY 60 million at that time [5]. In 2017, a bug in a smart contract library used by the Parity Multisig Wallet caused the loss of CNY 30 million and the freezing of CNY 150 million worth of ETC [6]. In 2018, attackers exploited an integer overflow vulnerability in the BEC smart contract to transfer an extremely large amount of tokens to malicious accounts, causing instantaneous evaporation of over CNY 900 million [7]. The above cases are not isolated, and security vulnerabilities of smart contacts are disclosed every year [8]. Even worse, with the population of decentralized finance (i.e., DeFi), attacks on smart contracts and financial losses are surging [9]. Hence, security vulnerability has become a critical issue in smart contract applications.

In contrast to other scenarios, the problem of security vulnerabilities in smart contracts faces more challenges, which can be summarized into three aspects. First, smart contracts are publicly available and hold increasing financial values, making them extremely tempting targets for attackers [10]. Second, although smart contracts can be obfuscated by only publishing the bytecode, it can still be decompiled for security analysis, which makes it a target of choice for attackers. Third, defects in smart contracts cannot be rectified due to the immutable nature of blockchain, which suggests the significant importance of vulnerability detection, especially before deployment on the blockchain.

Recently, many research efforts have been focused on detecting the security vulnerabilities of smart contracts [11–27], while most of them are inspired by or inherited from popular methods in conventional software testing. For example, fuzzing, an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program, is used to explore the state space of smart contract executions and thus detect deep vulnerabilities [11–14]. Symbolic execution, another software-testing technique that executes programs with symbolic inputs instead of concrete ones, is also applied to find vulnerabilities for smart contracts [15–19]. Both fuzzing and symbolic execution need smart contracts to be actually executed, which requires certain testing environments. In contrast to them, formal verification establishes mathematical models to determine the correctness of software behaviors before deployment. Owing to the usually small code size of smart contracts, formal verification methods are quite applicable to this scenario and thus are attracting growing attention from both academic fields and Solidity language committees [20–23]. Static analysis is another debugging method that examines the code without having to execute it. Program features or intermediate representations are extracted from smart contracts for vulnerability detection [24–27]. Among the above approaches, static analysis is considered a competitive and practical way to detect vulnerabilities in smart contracts since it does not necessitate actual code execution or complex formal specifications. However, learning more directional vulnerability features is a key factor to determine the effectiveness of static analysis tools. Most existing methods prefer to utilize abstract features and violation patterns to identify vulnerabilities [24–27], without taking full advantage of syntactic information and code structures.

In this paper, we build a graph representation of smart contracts, i.e., abstract semantic graph (ASG), based on a set of existing graph representations, such as abstract syntax tree (AST), control flow graph (CFG), and data flow graph (DFG). The basic idea of ASG is to mix up the above multi-views together into one single view such that more syntactic and semantic information can be preserved. In order to detect vulnerabilities of smart contracts in forms of both source code and bytecode, we propose two novel approaches called *ASGVulDetector* and *BASGVulDetector* through calculating the similarity between the code under test and the known vulnerable code via the graph matching network (GMN). Our approaches generally consist of three steps: Firstly, smart contracts are transformed to graph representations. Secondly, graph embedding is conducted to calculate vector representations for these contracts. Thirdly, similarities of each code pairs are measured through GMN. To fully utilize the abstract code structures, we construct an ASG by enriching the AST with control flow and data flow when the source code is available, while the ASG of a

smart contract in bytecode is extracted by decompiling and enriching basic block sequences with control flow. Based on that, we apply two training models, i.e., graph neural network (GNN) and GMN, on ASG to investigate the impact of different models. To validate the effectiveness of the proposed approaches, we conduct extensive experiments on a dataset of smart contracts, as well as comparing them with competitive vulnerability detection tools. The main contributions of this paper are summarized as follows.

- We design a novel graph representation ASG for smart contracts in forms of source code and bytecode, which takes advantage of both syntactic information and code structural features.
- We apply a graph-matching network model based on the ASG representation for vulnerability detection in smart contracts. We adopt two different training models, i.e., GNN and GMN, to analyze the difference between their performance.
- We implement two tools called *ASGVulDetector* and *BASGVulDetector* for smart contracts in source code and bytecode, respectively, and conduct extensive experiments to evaluate their effectiveness and superiority through comparisons with competitors.

The rest of this paper is organized as follows: Section 2 introduces the background of smart contracts, potential vulnerabilities, and graph-matching network. Section 3 briefly reviews existing approaches on the vulnerability detection of smart contracts. Section 4 elaborates details of the proposed approaches while Section 5 gives the experimental results. FInally, conclusions and suggestions on future work are provided in Section 6.

2. Background

2.1. Smart Contract in a Nutshell

Smart contracts are simply programs stored on a blockchain that execute when predefined conditions are met, as shown in Figure 1. They essentially automate the execution of an agreement between untrustworthy parties without the involvement of a trusted third party. There are various blockchain platforms that support smart contracts, among which Ethereum is the most common one [28]. It provides a decentralized ecosystem for smart contracts through a specific programming language called Solidity and a Turing-complete machine called EVM. The general workflow of a smart contract is as follows. First, source codes that represent terms of an agreement are written in a high-level language (e.g., Solidity) and then complied into bytecodes (e.g., EVM codes). Then, the bytecodes are uploaded to the blockchain in a form of a transaction, and stored in a block of the distributed ledger. Once the predetermined condition is satisfied, these bytecodes are translated into instructions or operation codes running on EVM to execute business logic.

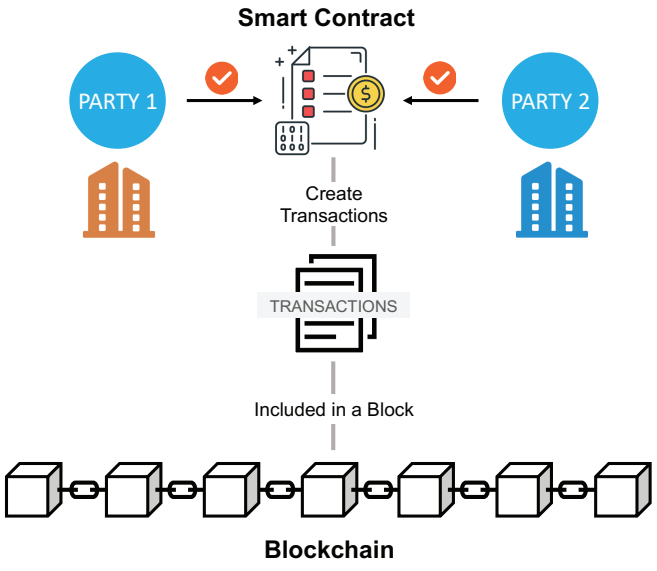


Figure 1. Blockchain-based smart contract.

Since smart contracts encapsulated in blockchain transactions are publicly accessible, code flaws can be analyzed and exploited by anyone, including attackers. Moreover, due to the immutable nature of blockchain, errors in smart contracts cannot be rectified once deployed. As the financial values held in the blockchain network are increasing, smart contracts have already become tempting targets. It is worth noting that both the source code and bytecode of smart contracts are choices of a target for attackers.

2.2. Four Types of Vulnerabilities in Smart Contracts

Smart contract platforms could encounter vulnerabilities at the blockchain level, EVM level, and contract level [11]. In this paper, we focus on contract-level vulnerabilities. Particularly, we consider four types of vulnerabilities that are related to code appearance and structures. More types and details of smart contract vulnerabilities can be found in related work [29,30].

2.2.1. Reentrancy

Reentrancy vulnerability happens when a built-in token transfer function, i.e., *call.value*, can call back to itself from an external invocation. In other words, such a function can be re-entered to perform unexpected token transfers.

Figure 2 shows an example code of reentrancy vulnerability. The attack occurs when a function makes an external call to another untrusted contract (e.g., *msg.sender.call.value* at line 7). Then, the untrusted contract can make a recursive call back to the original function (e.g., *withdraw*()) to drain funds. The underlying cause of reentrancy vulnerability is that the vulnerable contract allows external calls to take over the control flow and make unexpected changes to internal variables. The notorious DAO attack is the most famous example of a reentrancy hack [5].

```
1  contract Simple {
2     mapping (address => uint) public userBalance;
3     function withdraw () public {
4        uint amount;
5        amount = userBalance [msg.sender];
6        if (amount > 0){
7           msg.sender.call.value (amount)();
8           userBalance[msg.sender] = 0 ;
9        }
10    }
11 }
```

Figure 2. An example smart contract with reentrancy vulnerability.

2.2.2. Timestamp Dependency

Timestamp-dependency vulnerability happens when smart contracts utilize the value of *block.timestamp*, which is a block built-in variable, as a part of the call condition to execute critical operations (e.g., token transfer). The value of *block.timestamp* is generated by the node executing the smart contract (e.g., anonymous miners), which makes it manipulable and vulnerable to attacks.

2.2.3. Block info Dependency

Similar to the timestamp-dependency vulnerability, block-info-dependency vulnerability happens when the execution of a smart contract relies on block-related variables, such as *block.number*, *block.hash*, etc. However, since these values are more predictable due to the blockchain protocol, block-related variables can be manipulated to perform unexpected behaviors.

2.2.4. Tx.Origin Attack

Tx.Origin is a global variable that refers to the address of the account that launches the transaction. The Tx.Origin attack could happen when the variable *Tx.Origin* is used to authorize calls from external accounts (e.g., $Tx.Origin == msg.sender$). This vulnerability can be exploited jointly with the phishing attack to drain a contract of all funds.

2.3. Graph-Matching Network

Graph-structured objects are widely encountered in many real-world applications. In the past few years, GNNs have been an effective class of deep learning methods for learning the representations as well as performing supervised prediction based on graph-structured objects, such as social media network [31], vehicular network [32], and program code structures [27]. GMN is an extension to GNNs for the purpose of graph similarity learning [33]. Instead of computing graph representations independently for each graph, GMNs take a pair of graphs as input and compute a similarity score by a cross-graph attention mechanism at the cost of certain computation efficiency.

3. Related Work

Extensive studies have been focused on vulnerability detection in smart contracts. According to the software-testing technique used to reach the goals, existing approaches can be classified into four categories: fuzzing, symbolic execution, formal verification, and static analysis.

Fuzzing is an automated software testing technique that feeds the program under test with a large volume of random inputs to identify code errors and security vulnerabilities from black, grey or white box perspectives. Jiang et al. [11] proposed the first fuzzing framework called *ContractFuzzer* for detecting security vulnerabilities in smart contracts on the Ethereum platform. *ContractFuzzer* generates fuzzing inputs according to the ABI specifications of smart contracts, and instruments EVM to provide feedback during the execution. However, *ContractFuzzer* only supports a custom set of built-in detectors. In contrast to that, Grieco et al. [12] implemented another Ethereum smart contract fuzzer called *Echidna*, which supports a large set of features based on experience with security audits, such as custom property checking, assertion checking, and estimation of maximum gas usage. Inspired by the well-known fuzzer for C programs, i.e., AFL, Nguyen et al. [14] proposed an adaptive fuzzer called *sFuzz* for Ethereum smart contracts. It combines the strategy in AFL and a lightweight multi-objective adaptive strategy targeting those hard-to-reach branches. To explore more transition states and thereby detect deep vulnerabilities, Wüstholz et al. [13] designed a greybox fuzzer for smart contracts called *Harvey*, which is equipped with two key techniques, i.e., input prediction and demand-driven sequence fuzzing.

Symbolic execution is a way of executing a program abstractly so that one abstract execution covers multiple possible inputs that share the same program path. It has been widely used to explore the program states and detect security vulnerabilities for smart contracts. Luu et al. [15] built a symbolic execution tool called *Oyente* for Ethereum smart contracts, which takes bytecodes as input and consists of four main components to perform CFG construction, symbolic execution, constraint solving, and false alarm filtering. On the basis of *Oyente*, Torres et al. [16] proposed a symbolic execution framework combined with taint analysis called *Osiris*, focusing on accurately detecting integer vulnerabilities in smart contracts of the Ethereum platform. Mossberg et al. [17] implemented a dynamic symbolic execution framework called *Manticore* for analyzing binaries of Ethereum smart contracts. Its flexible architecture allows performing symbolic execution on alternative platforms. To effectively find vulnerable transaction sequences in smart contracts, So et al. [18] presented a symbolic execution technique called *Smartest*, which utilizes statistical language models to guide symbolic execution so that it can effectively prioritize program paths that are likely to reveal vulnerabilities. In view of the fact that most smart contract symbolic execution tools perform analysis on bytecode, Lin et al. [19] designed a source-level symbolic

execution for Ethereum smart contracts, which can take advantage of high-level semantic information in the source code.

Formal verification conducts mathematical analysis to prove or disprove the correctness of a program by checking its formal model against a certain formal specification. Tsankov et al. [21] presented a formal verification tool called *Securify* to detect vulnerabilities in smart contracts. It is integrated with compliance and violation patterns described in domain-specific language to examine whether there are loopholes in the contracts. Bai et al. [20] established the formal model of smart contracts and utilized model checking to verify the correctness as well as important properties of the contracts. Albert et al. [22] proposed a formal verification tool called *SAFEVM*, which consists in decompiling the Ethereum bytecode into a C program with ERROR annotations so that existing verification engines developed for C programs can be used to verify these bytecodes. Antonino et al. [23] designed a bounded model checker for Solidity called *Solidifer*, which leverages Boogie, an intermediate verification language, to capture accurate semantics of Solidity's memory model, lazy contract deployment and memory precise verification harness.

Static analysis is a method of computer program debugging by examining the source code without executing the program, which can guarantee full code coverage and fast detection efficiency. Tikhomirov et al. [24] proposed an extensible static analysis tool called *SmartCheck* for Ethereum smart contracts, which converts Solidity code into XML-based intermediate representation and checks it against XPath patterns derived from vulnerable behaviors. Similarly, Feist et al. [25] represented another static analysis tool called *Slither*, which translates Solidity code into a user-defined intermediate representation called SlitherIR. Moreover, it combines data flow and taint analysis techniques to extract more semantic information and detect pattern violations. Xue et al. [26] summarized existing strategies to avoid reentrancy vulnerability as path protection techniques (PPTs), and proposed a static analysis tool called *Clairvoyance* for reentrancy bug detection. It relies on a cross-contract inter-procedural CFG (ICFG) representation and PPT patterns to identify potential paths that contain reentrancy. As the most similar work to our approaches, Zhuang et al. [27] constructed a contract graph to represent both syntactic and semantic structures of a smart contract function. Based on that, the graph neural network and temporal message propagation network are separately introduced as the training models for vulnerability detection. However, the graph defined in this work involves three types of nodes with complex edge information, which can fail to generalize across different contracts.

Although the above studies provide promising approaches for vulnerability detection in smart contracts, there still remain limitations and challenges. In general, fuzzing and symbolic execution have to execute the smart contracts, either concretely or symbolically, which require sandboxed environments and have execution overhead. Formal verification-based techniques are limited by the rarely available formal specification of built-in functions. Static analysis is an effective way to find bugs, but usually relies on the accuracy of an intermediate representation to feature the code. Furthermore, patterns or models for detecting bugs are of key importance to the performance of vulnerability detection. Most existing static analysis methods require source code to perform vulnerability detection. In this paper, we attempt to address these issues through a novel graph representation for both source code and bytecode, coupled with the graph-matching network technique.

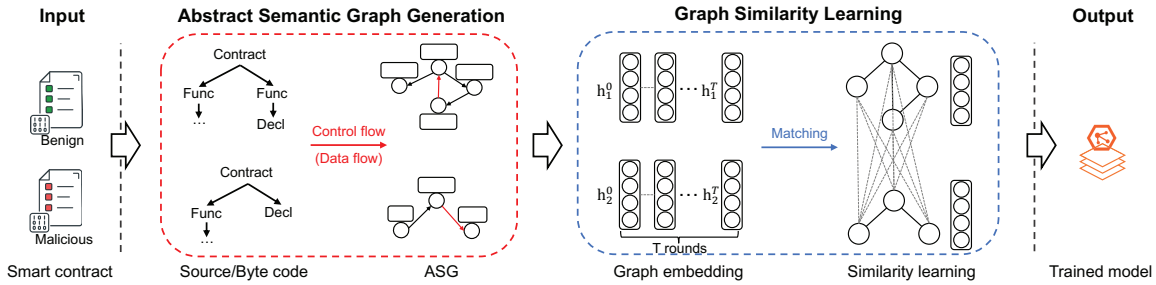
4. Methodology

This section elaborates details of the proposed methodology, which mainly consists of graph generation and similarity learning.

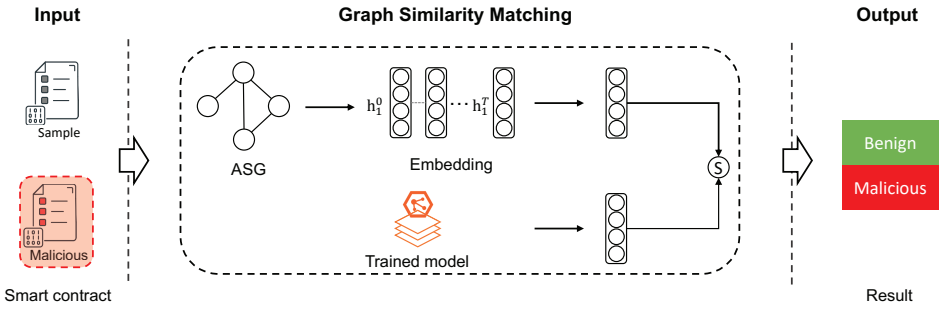
4.1. Framework

The underlying idea of our approaches is that ASG, extended syntactic information with more code structural features, is expected to preserve more syntactic and semantic information. Furthermore, GMN has been proved to be an effective model for matching graph-structured objects [33]. Figure 3 illustrates the general framework of the proposed

approaches, which is comprised of two main phases, including model training and vulnerability prediction.



(**a**)



(**b**)

Figure 3. The general framework of our proposed approaches. (**a**) Model training (**b**) Vulnerability prediction.

As shown in Figure 3a, given a dataset that contains both benign and malicious contracts, the workflow of model training can be further divided into two steps. The first step is to generate ASG from the code. When the source code is available, ASG is based on the AST but enriched by control flow and data flow. Otherwise, it consists of decompiling and augmenting basic block sequences with control flow. The second step is to calculate vector representations for smart contracts using GNN, then utilize GMN to learn similarities between the graph pairs. Note that our two proposed tools *ASGVulDetector* and *BASGVulDetector* only differ in the graph generation step; the other following procedures are performed in the same manner.

To reduce the false positive rate of detection, each type of vulnerability can be isolated by an independent trained model. Nevertheless, each type of vulnerability is predicted through the same procedure, as shown in Figure 3b. First, suspicious vulnerability types of trained models should be selected before the sample contract is fed to the tool. Then, the sample contract is processed as a fresh input to the training model, which means procedures of ASG generation, graph embedding will be conducted to obtain a vector representation of the sample contract. Finally, such a vector is evaluated in the trained model in terms of similarity with known vulnerabilities. In order to detect different types of vulnerabilities, the sample contract will be evaluated through different trained models in series, parallel, or any combination thereof.

4.2. Abstract Semantic Graph Generation

Graph representation has more capabilities to preserve the semantic features of a program [34]. There already exists plenty of sets of graph representations, including AST, CFG, and DFG, which are derived from different aspects, such as syntax, control flow, and data flow, separately. AST provides the abstract syntactic structure of the source code, where leaf nodes represent operands and non-leaf nodes represent operators. This is quite useful to figure out key syntactic features. For example, if a smart contract contains *call.value*, it can

be regarded as a candidate of reentrancy vulnerability. However, in most cases, control flow and data flow, which represent the order of statements and the processing flow of variables, respectively, are more likely to reveal program behaviors at runtime. The underlying idea of ASG is to combine the above multi-views together into one single view such that more syntactic and semantic information can be preserved in this novel kind of intermediate representation for smart contracts. In this way, it is more feasible and efficient to use one single graph instead of multiple graphs to train the neural network model.

4.2.1. ASG from Source Code

Given the source code in Solidity of a smart contract, it can be parsed into AST by a mature set of compiler tools, e.g., SolidityParser and solc-ast. The top part of Figure 4 illustrates an example AST parsed from the *Simple* contract in Figure 1.

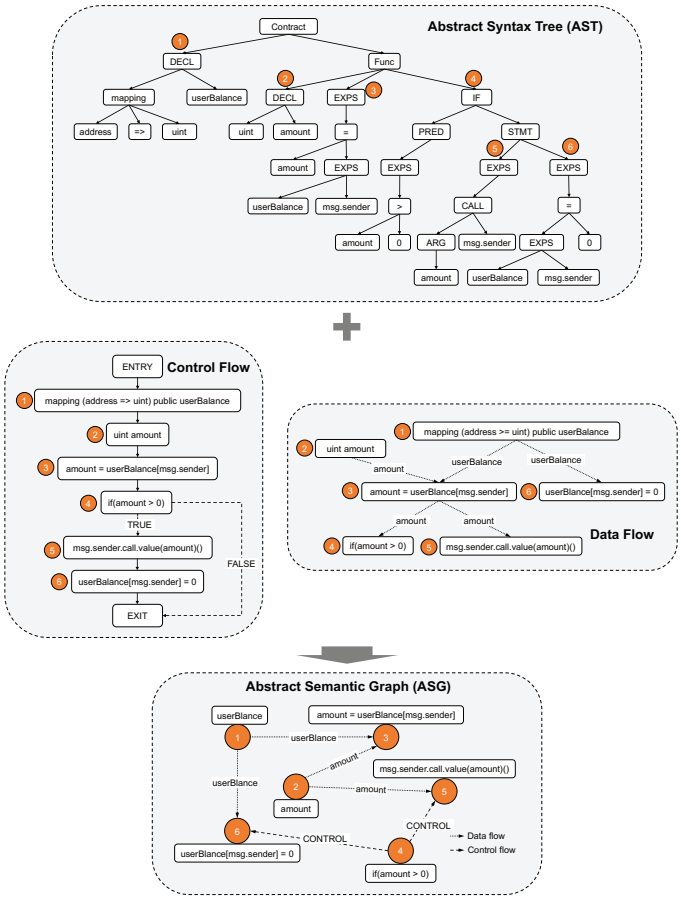


Figure 4. Abstract semantic graph generation based on abstract syntax tree and enriched by control flow and data flow.

It can be observed that AST provides a fine-grained syntactic representation of the source code. However, with the scale of the program increasing, its AST would be extremely complicated. Directly leveraging AST as the basis will largely increase the numbers of nodes and edges in ASG, which can result in intractable problems such as gradient exploding in the follow-up model training phase. In this case, ASG takes a statement-level granularity, which is a single line of code-based elements, as annotated by orange circled numbers in Figure 4, to construct the dedicated graph. Formally, ASG is an abstract semantic graph $G = (V, E)$, where V is a set of nodes and E is a set of edges in the graph. Each node $v_i \in V$ represents a statement of the source code, while an edge $e_{i,j} \in E$ indicates that there exists either a control dependency (in control flow) or a data dependency (in data flow) from node v_i to v_j.

Control flow reflects the orders of statements in the smart contract. Similar to other programming language, Solidity uses keywords such as **if**, **while**, **for**, **continue**, and **break** to control statement orders; rest statements without these keywords are executed consecutively. The control flow can also be extracted from the source code, and thus can be introduced into the ASG accordingly. Note that each node of both ASG and CFG represents the same statement of the contract. Formally, a direct edge $e_{i,j} =< v_i, v_j >$ indicates that statement v_j is executed after the completion of statement v_i. Figure 4 also gives an example control flow for the *Simple* contract, where most of control edges are sequential and only one **if** statement can change the execution order. In a contract that involves complex business logic, its control flow can be more sophisticated.

Data flow is another orthogonal representation of the source code, which describes data dependencies between a number of operations. However, data dependencies rely on the execution orders of the statements. Hence, the search of data dependencies has to consider every control path that leads to the endpoint of the data dependency. Formally, a direct edge $e_{i,j} =< v_i, v_j >$ indicates that data produced by statement v_i are consumed by statement v_j. An example data flow for the *Simple* contract is also demonstrated in Figure 4. There are two variables, i.e., *amount* and *userBalance*, defined and used in the smart contact. It can be investigated that variable *userBalance* in v_3 is related to the **if** statement in v_4 through variable *amount*. Such a relationship is quite meaningful for analyzing program behaviors, yet can hardly be obtained from AST or control flow.

The overall workflow of ASG generation is depicted by Algorithm 1. The source code of a given smart contract S is parsed to its AST (Line 1), which is used to initialize ASG in statement-level granularity (Line 2). Then, AST is traversed with awareness of control flow structure and variable definitions (Lines 4–13). Note that control edges can be pointed out with the aid of CFG provided by a Solidity compiler tool (Lines 5–9), while variables defined in the program are recorded in an individual set *VariableSet* simultaneously (Lines 10–12). After the control edges are all determined, there will be a set of control paths in the ASG. Thus, data edges are appended to the ASG by examining every variable along each control path (Lines 14–18). With the completion of all above procedures, the ASG for the given smart contract is obtained finally. An example ASG of the *Simple* contract is illustrated at the bottom of Figure 4. Note that sequential edges from the control flow are omitted in the example ASG since the sequence number of each node is self-explained.

Algorithm 1 ASG generation from source code.

Input: Source code of smart contract, S
Output: Abstract semantic graph, ASG

```
1:  Parse the source code of S to an abstract syntax tree AST
2:  Initialize ASG with AST in statement-level granularity
3:  VariableSet←∅
4:  for each node v in AST do                          ▷ Add edges of control flow
5:      if v is in the control flow then               ▷ such as for, while, if, etc.
6:          Add control edge(s) to ASG
7:      else
8:          Add sequential edge(s) to ASG
9:      end if
10:     if v is a variable definition then
11:         Add var(s) in statement v to the set of VariableSet
12:     end if
13: end for
14: for each var in VariableSet do                     ▷ Add edges of data flow
15:     if var is in a control path of ASG then
16:         Add data edge(s) between each pair of nodes that both contain var
17:     end if
18: end for
19: return ASG
```

4.2.2. ASG from Bytecode

Recently, many smart contracts have been uploaded to the blockchain in forms of bytecode without publishing the source code. In this case, generating ASG from bytecode is much more challenging [35], but they share the same methodology, targeting to extract syntactic and semantic features to represent the code. Given the bytecode of a smart contract, we set up three steps to generate ASG from the bytecode, including decompilation, CFG construction, and normalization. For the purpose of clarity, we use BASG (Bytecode ASG) to denote ASG from bytecode in this section. Figure 5 illustrates an example of BASG, which is generated based on basic block sequences and enriched by control flow.

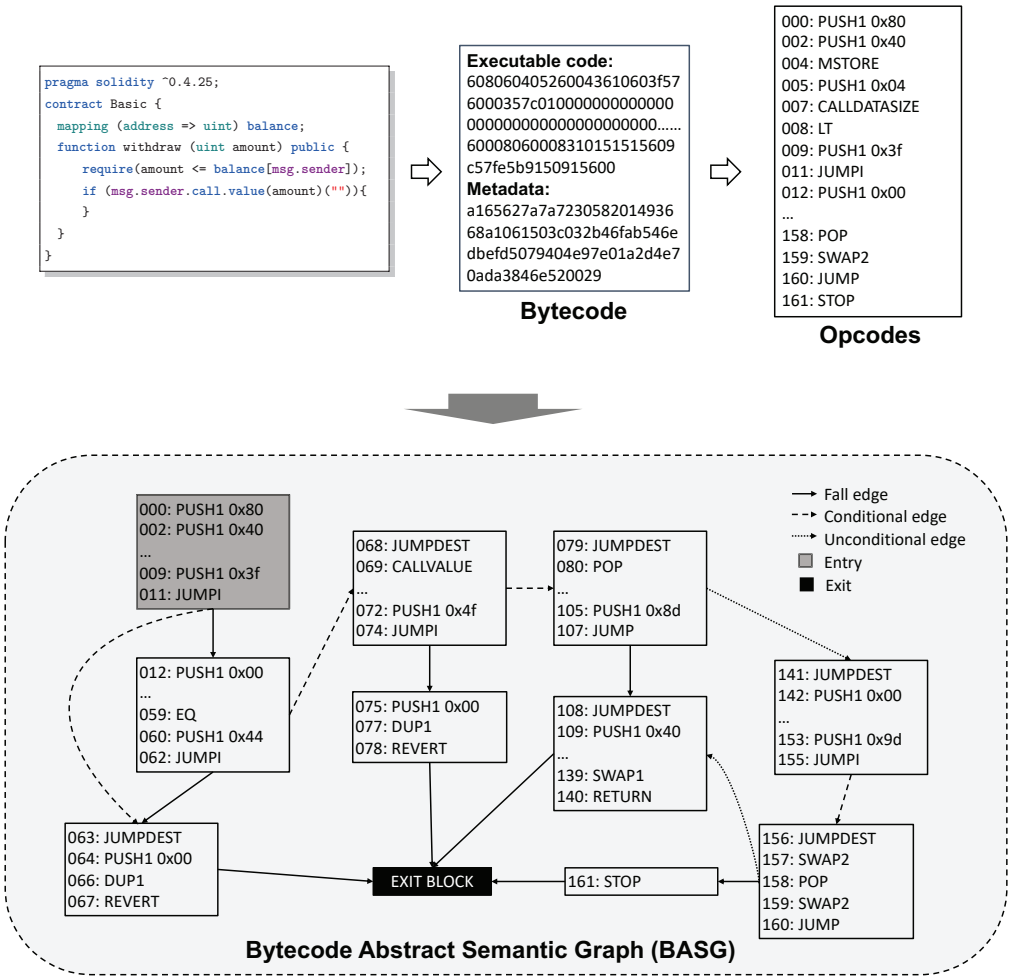


Figure 5. Bytecode abstract semantic graph generation based on basic block sequences and enriched by control flow.

Bytecode is a string of hexadecimal numbers; in order to extract syntactic information from the bytecode, it should be decompiled the first time. The bytecode deployed on the blockchain, i.e., runtime code, can be divided into three code segments. The first segment contains the opcodes that the EVM executes. The second segment is optional, and usually used to store static data. The last segment contains metadata, such as hashes of the code and compiler version. In Figure 4, an example of bytecode is also provided. According to the yellow paper of Ethereum [36], we can easily obtain opcodes from the bytecode. For instance, the bytecode 0×6070604001 can be decompiled to (PUSH1, 0×70, PUSH1, 0×40, ADD), where opcode 0×60 indicates PUSH1 with one operand (i.e., 0×70 and 0×40) and opcode 0×01 indicates ADD. Thus, EVM will execute two PUSH1 instructions to push two values, i.e., 0×70 and 0×40, to the stack, then consume these two elements

to execute an ADD instruction, leaving their sum 0 × B0 as a result. By traversing all of the runtime code, the opcodes of the overall smart contract can be obtained.

Although opcodes provide elementary information about the bytecode, they are still not quite analyzable. In order to obtain more semantic information and thus to understand the program behaviors, these opcodes should be analyzed in an organized form. In this case, we construct CFG to further represent the code as follows. First, the opcodes are grouped in basic blocks, which are also utilized as nodes of BASG. Then, opcodes that alter the control flow of the program (e.g., JUMP and RETURN) are resolved to augment control edges to the BASG. A basic block is a sequence of opcodes that are executed consecutively without any instruction that alters the flow of control. In Solidity, opcodes that can alter the control flow are mainly listed as JUMP, JUMPI, STOP, REVERT, RETURN, INVALID, and SELFDESTRUCT, but for jump-like instructions, the destination needs to be resolved from the stack. We employ the symbolic stack execution technique [35] to obtain these control edges. Accurate CFG construction for the bytecode is still an ongoing research topic which is beyond the scope of this paper.

Before the current form of BASG is fed to the training model, it is required to perform normalization to mitigate syntax noises, thus improving the semantic expressiveness of the BASG. We consider to normalize both opcodes and operands in this step. On one hand, opcodes with the same functionality, such as PUSH1 and PUSH32, are normalized to a general form, such as PUSHX. Stack operators that manipulate elements internally, such as DUP, POP, and SWAP, can be omitted from the semantic perspective. Table 1 lists all mappings from original opcodes to their normalized forms. On the other hand, in bytecode, operands that follow opcodes are usually hexadecimal numbers, which are determined by the program layout. Directly leveraging them as features will introduce noises in the training model, whereas replacing them with uniform notations has little influence on featuring the code. The operand normalization mappings used in this paper are provided in Table 2, where we classify the opcodes into six categories according to the types of their operands, i.e., arithmetic, block, logic, memory, store, and bit.

Table 1. Opcode normalization.

Opcode	Normalized Code
LOG0-LOG4	LOGX
PUSH1-PUSH32	PUSHX
DUP1-DUP16	-
SWAP1-SWAP16	-
POP	-

Table 2. Operand normalization.

Opcode	Normalized Operand
ADD, MUL, SUB, EXP, SIGNEXTEND	ArithData
BLOCKHASH, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT	BlockData
LT, GT, SLT, SGT, EQ, ISZERO	LogicData
MLOAD	MemData
SLOAD	StorData
BYTE, SHL, SHR, SAR AND, OR, XOR, NOT	BitData

The overall workflow of BASG generation is described by Algorithm 2. The bytecode of a smart contract B is decompiled to opcodes at first time (Line 1), which are grouped in basic blocks according to the opcodes alerting the flow of control (Lines 2–10). Each basic block is regarded as a node of the $BASG$. Then, control edges consisting of fall edges, conditional edges, and unconditional edges are augmented to the $BASG$ (Lines 11–13).

Finally, opcodes and operands are normalized to their general forms following the rules as aforementioned (Lines 14–17).

Algorithm 2 BASG generation from bytecode.

Input: Bytecode of smart contract, B
Output: Bytecode abstract semantic graph, $BASG$

```
1: Decompiling B to extract Opcodes
2: BB←∅
3: for op in Opcodes do                                   ▷ Obtain basic block
4:     if op is [JUMP JUMPI STOP REVENT RETURN INVALID SELFDESTRUCT] then
5:         BASG.Node.add(BB)                              ▷ Add current basic block
6:         BB←∅
7:     else
8:         BB.add(op)                                     ▷ Add current Opcode
9:     end if
10: end for
11: BASG.Edge.addFallEdges()
12: BASG.Edge.addConditionalEdges()
13: BASG.Edge.addUnconditionalEdges()
14: for BB in BASG.Node do                                ▷ Normalization
15:     NormalizeOpcode(BB.Opcode)
16:     NormalizeOperand(BB.Operand)
17: end for
```

4.3. Graph Similarity Learning

Compared to traditional deep neural network models such as the convolutional neural network (CNN) and recurrent neural network (RNN), the graph neural network (GNN) is more applicable to handle graph structured data, whereas the graph matching network (GMN) enhances it through a cross-graph attention-based matching mechanism. In this paper, we use a GNN for graph embedding, and jointly apply a GMN to learn the similarity between each pair of contract graphs.

4.3.1. Contract Graph Embedding

So far, we transformed a smart contract to its graph representation. In ASG, each node represents a statement of the source code, while each edge represents a control or data dependency in the code. In BASG, each node represents a basic block of the bytecode, while each edge represents a control dependency. Although such a contract graph is able to represent plentiful syntactic and semantic information of the program, it is usually in a high-dimension space and can hardly be processed by a deep neural network. Thus, it has to be further converted to a vector representation, which is formally called embedding.

We use a gated graph neural network (GGNN) for graph embedding. Given a contract graph $G = (V, E)$, $\mathbf{x}_i$ and $\mathbf{x}_{i,j}$ are feature vectors of node v_i and edge $e_{i,j}$, respectively. Then, these features are encoded through separate multi-layer perceptrons (MLPs):

$$\begin{aligned} \mathbf{h}_i^{(0)} &= \mathrm{MLP}_{\mathrm{node}}(\mathbf{x}_i), \quad \forall i \in V \\ \mathbf{e}_{ij} &= \mathrm{MLP}_{\mathrm{edge}}(\mathbf{x}_{ij}), \quad \forall (i,j) \in E, \end{aligned} \tag{1}$$

where $\mathbf{h}_i^{(0)}$ is the embedding of node v_i in the initial state (state 0), and $\mathbf{e}_{ij}$ is the embedding assigned to edge $e_{i,j}$. The hidden state of nodes are updated in the message passing step by

$$\begin{aligned} \mathbf{m}_{j \to i} &= f_{\mathrm{message}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}, \mathbf{e}_{ij}\right), \quad \forall (i,j) \in E \\ \mathbf{h}_i^{(t+1)} &= f_{\mathrm{update}}(\mathbf{h}_i^{(t)}, \sum_j \mathbf{m}_{j \to i}), \end{aligned} \tag{2}$$

where f_{message} is a message function that collects messages from node v_i's neighborhoods, and f_{update} is a node update function that combines these messages and the last hidden state of node embedding. In this paper, we use an MLP and a gated recurrent unit (GRU) for the two functions, respectively. At each iteration t of the GNN, $\sum_j \mathbf{m}_{j\to i}$ aggregates the set of embeddings of v_i's neighborhoods through a simple *sum* function. Based on that, f_{update} generates the updated embedding $\mathbf{h}_i^{(t+1)}$ for each node. After T iterations of the GNN message passing, a set of embeddings of all nodes are obtained and denoted as $\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V}$. To compute the graph-level representation for a contract, we follow the readout function proposed in [33]:

$$\mathbf{h}_G = f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V}\right) = \text{MLP}_G\left(\sum_{i\in V}\sigma\left(\text{MLP}_{\text{gate}}\left(\mathbf{h}_i^{(T)}\right)\right)\odot\text{MLP}\left(\mathbf{h}_i^{(T)}\right)\right), \tag{3}$$

which leverages the weighted sum with gating vectors to aggregate across nodes. Note that given the graph representations $\mathbf{h}_{G_1}$ and $\mathbf{h}_{G_2}$ for two graphs G_1 and G_2, their similarity can be calculated using metrics such as Euclidean, Hamming, or cosine similarities, and thus can already be used for vulnerability detection, which will be investigated in Section 5.5.

4.3.2. Graph Matching Network

Rather than separately mapping each contract graph to a vector in GNN, GMN takes a pair of contract graphs at a time and learns the embeddings through cross-graph attention-based matching. Initially, the encoding step of GMN is the same as that of GNN, as shown in Equation (1). The main difference between the two models lies on the propagation process, which can be described as the following:

$$\begin{aligned}
\mathbf{m}_{j\to i} &= f_{\text{message}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}, \mathbf{e}_{ij}\right), \quad \forall (i,j)\in E_1\cup E_2\\
\boldsymbol{\mu}_{j\to i} &= f_{\text{match}}\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right), \quad \forall i\in V_1, j\in V_2, \text{ or } i\in V_2, j\in V_1\\
\mathbf{h}_i^{(t+1)} &= f_{\text{update}}\left(\mathbf{h}_i^{(t)}, \sum_j \mathbf{m}_{j\to i}, \sum_{j'}\boldsymbol{\mu}_{j'\to i}\right),
\end{aligned} \tag{4}$$

where we still use an MLP and a GRU for f_{message} and f_{update}, respectively, and f_{match} is introduced to exchange cross-graph information through

$$\begin{aligned}
a_{j\to i} &= \frac{\exp\left(s_h\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right)\right)}{\sum_{j'}\exp\left(s_h\left(\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}\right)\right)}\\
\boldsymbol{\mu}_{j\to i} &= a_{j\to i}\left(\mathbf{h}_i^{(t)} - \mathbf{h}_j^{(t)}\right),
\end{aligned} \tag{5}$$

with s_h as a vector similarity metric and $a_{j\to i}$ as the attention weight. In this paper, we use cosine similarity for s_h. It can be observed that the updater $\mathbf{h}_i^{(t+1)}$ of GMN not only considers the last hidden state and the aggregated message, but also takes $\sum_{j'}\boldsymbol{\mu}_{j'\to i}$ as input, which measures the differences between $\mathbf{h}_i^{(t)}$ and all its neighborhoods. For the readout function of GMN, we still follow Equation (3), and the similarity of a graph pair can be computed by

$$\begin{aligned}
\mathbf{h}_{G_1} &= f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V_1}\right)\\
\mathbf{h}_{G_2} &= f_G\left(\left\{\mathbf{h}_i^{(T)}\right\}_{i\in V_2}\right)\\
s &= f_s\left(\mathbf{h}_{G_1}, \mathbf{h}_{G_2}\right),
\end{aligned} \tag{6}$$

where s is a similarity score and f_s is a similarity metric.

According to the above setting, most parts of our GMN model are similar to those of the GGNN model, while differences mainly occur at the propagation stage. We will evaluate the effectiveness of GMN on improving the performance of vulnerability detection in smart contracts later.

5. Evaluation

This section evaluates *ASGVulDetector* and *BASGVulDetector* by comparing it with competitive tools in terms of performance and efficiency, while the GMN model used in the proposed approaches is also investigated extensively.

5.1. Experimental Setup

We implemented prototypes of *ASGVulDetector* and *BASGVulDetector* in Python. The AST of a smart contract in source code was parsed by solc-ast (https://github.com/iamdefinitelyahuman/py-solc-ast accessed on 21 July 2021), while the machine learning models were powered by the PyTorch library (https://github.com/pytorch/pytorch accessed on 18 May 2021). The computer we used for model training and vulnerability detection was a desktop with Intel Core i7 CPU, 16 GB memory, and NVIDIA Tesla T4 GPU, running on Ubuntu 20.04 operating system. We collected a dataset of 5735 open-source smart contracts from SmartBug [37], SolidFI [38], and other labeled contracts, which contains the four types of vulnerabilities, i.e., reentrancy, timestamp dependency, block info dependency, and Tx.Orgin. These labeled contracts were randomly divided into a training set and a testing set, with a 70–30 split. In experiments for *BASGVulDetector*, these smart contracts were compiled to bytecodes beforehand. The threshold of similarity for the GMN model to detect vulnerability was set to 0.85, and other parameters of training models were determined according to the experiment conducted in Section 5.4.

Since our two approaches support different scenarios, i.e., source code and bytecode, we evaluated them separately. For the purpose of comparisons, in experiments for *ASGVulDetector*, we selected three state-of-the-art vulnerability detection tools, i.e., *SmartCheck* [24], *Slither* [25], and *DR-GCN* [27], all of which belong to the static analysis category and are applicable for the source-code scenario. As aforementioned, *SmartCheck* converts Solidity code to a XML-based intermediate representation, and detects vulnerability by checking XPath patterns. *Slither* follows the same idea but defines its own intermediate representation, i.e., SlitherIR, and combines data flow and taint tracking to analysis the code. Among the above competitors, *DR-GCN* is the most similar work to our approaches. It constructs a graph with three types of nodes and four types of edges, and uses GNN for vulnerability detection. In experiments for *BASGVulDetector*, we selected three other vulnerability detection tools, i.e., *ContractFuzzer* [11], *Oyente* [15], and *Securify* [21], which support bytecode and belong to the categories of fuzzing, symbolic execution, and formal verification, respectively. *ContractFuzzer* [11] instruments EVM and performs fuzz testing on smart contracts. *Oyente* [15] formalizes the semantics of Ethereum smart contracts and provides a symbolic execution tool to detect bugs. *Securify* [21] describes all compliance and violation patterns in a designated domain-specific language, and conducts formal verification on smart contracts. Comparisons with the above methods are expected to validate the effectiveness and advantages of our approaches.

5.2. Performance Evaluation

In the first experiment, we investigated how well the proposed approaches could detect smart contract vulnerabilities. We compared *ASGVulDetector* and *BASGVulDetector* with their competitors in terms of accuracy (**Acc**), recall (**Rec**), precision (**Pre**), and F1-score (**F1**). The performance results of different methods are listed in Tables 3 and 4, respectively, from which we have the following observations.

Table 3. Performance comparison for *ASGVulDetector* ('-' denotes not applicable).

Vulnerability	Method	Acc (%)	Rec (%)	Pre (%)	F1 (%)
Reentrancy	SmartCheck	55.70	74.95	62.08	67.91
	Slither	70.51	85.93	75.83	80.57
	DR-GCN	66.21	81.78	71.08	76.06
	ASGVulDetector	**84.96**	**95.37**	**84.17**	**89.42**
Timestamp dependency	SmartCheck	40.41	77.44	37.45	50.49
	Slither	74.19	89.47	77.27	82.93
	DR-GCN	50.59	84.22	47.55	60.78
	ASGVulDetector	**87.02**	**94.59**	**89.09**	**91.76**
Block info dependency	SmartCheck	-	-	-	-
	Slither	67.77	84.46	70.43	76.81
	DR-GCN	-	-	-	-
	ASGVulDetector	**88.99**	**94.01**	**91.30**	**92.63**
Tx.Origin	SmartCheck	41.12	69.34	43.69	53.60
	Slither	60.96	82.09	62.97	71.27
	DR-GCN	-	-	-	-
	ASGVulDetector	**81.18**	**87.58**	**88.35**	**87.97**

Table 4. Performance comparison for *BASGVulDetector* ('-' denotes not applicable).

Vulnerability	Method	Acc (%)	Rec (%)	Pre (%)	F1 (%)
Reentrancy	ContractFuzzer	37.94	67.12	31.36	42.75
	Oyente	41.64	74.02	42.50	54.00
	Securify	53.62	77.46	54.42	63.93
	BASGVulDetector	**80.59**	**92.73**	**80.00**	**85.90**
Timestamp dependency	ContractFuzzer	32.96	82.94	28.36	40.70
	Oyente	43.24	76.62	38.45	52.55
	Securify	52.06	84.62	50.00	62.86
	BASGVulDetector	**79.28**	**92.17**	**81.36**	**86.43**
Block info dependency	ContractFuzzer	30.52	58.08	28.75	38.46
	Oyente	-	-	-	-
	Securify	49.40	72.15	53.75	61.60
	BASGVulDetector	**85.04**	**93.66**	**86.09**	**89.71**
Tx.Origin	ContractFuzzer	-	-	-	-
	Oyente	-	-	-	-
	Securify	47.12	71.93	51.00	59.68
	BASGVulDetector	**81.56**	**86.94**	**89.81**	**88.35**

It can be observed from Table 3 that *SmartCheck* commonly has the lowest **Acc** and **F1** among the four tools, e.g., only 55.7% and 67.91% for reentrancy. This phenomenon can be explained by the XML-based mechanism used in *SmartCheck*, which is utilized not only for intermediate representation but also to define violation patterns. Thus, the performance of vulnerability detection is limited by the expressiveness of XML language, and thus cannot reflect sufficient syntactic and semantic information. In contrast, *Slither*, as another tool based on intermediate code, provides much better detection performance than *SmartCheck*. This could profit from two intensive designs. One is that *Slither* uses a fine-grained intermediate form called static single assignment (SSA) to preserve more semantic information, while the other is that it employs data flow and taint tracking to obtain more understandings about program behaviors. *DR-GCN* has a relatively low detection performance, even worse than *Slither*. This can be explained by *DR-GCN* transforming code fragments of a contract to a graph that contains specific types of nodes and edges, which reduces the ability of representing smart contracts. In the cases of block info dependency and Tx.Origin, the contract code can even hardly be represented in the required form, which reveals that *DR-GCN* is applicable to a limited set of scenarios. Among the

four methods, *ASGVulDetector* performs the best in all cases. Specifically, *ASGVulDetector* improves the most of the three competitors in the four types of vulnerabilities on F1 score by 8.9, 8.8, 15.8 and 16.7 percentages, respectively. These improvements reveal the effectiveness of the enriched graph-based representation as well as the classification power of the graph-matching network.

As shown in Table 4, *ContractFuzzer* almost has the lowest detection performance in all cases. In the case of Tx.Origin, it even cannot find vulnerabilities in reasonable time. This is due to the fact that the fuzzing technique takes random inputs to trigger bugs, and thus its performance heavily relies on code coverage and testing time. *Oyente*, as a symbolic execution tool, is superior to *ContractFuzzer* since it can resolve code branches by constraint solving. However, it is still limited in complex branches, such as resolving a concrete value for a condition. In contrast to the above two methods, *Securify* is more stable and has better performance in all cases. This benefits from the formal methods used in *Securify*, but formal specifications covering all Solidity functions still remain challenging. *BASGVulDetector* wins the three competitors in all cases, improving *Security* in the four types of vulnerabilities on F1 score by 22.0, 23.6, 28.1, and 28.7 percentages, respectively. These improvements are contributed by our abstract representation of the bytecode, as well as the power of the GMN model.

5.3. Detection Time Cost

In the second experiment, we investigated the time cost of each method spent on vulnerability detection. Note that the preparation time for detection, including pattern extraction in *SmartCheck*, *Slither*, and *Securify*, model training in *DR-GCN* and our approaches, was not considered. We only took into account the detection time spent on each type of vulnerability. The average values for our two approaches are illustrated in Figures 6 and 7, respectively.

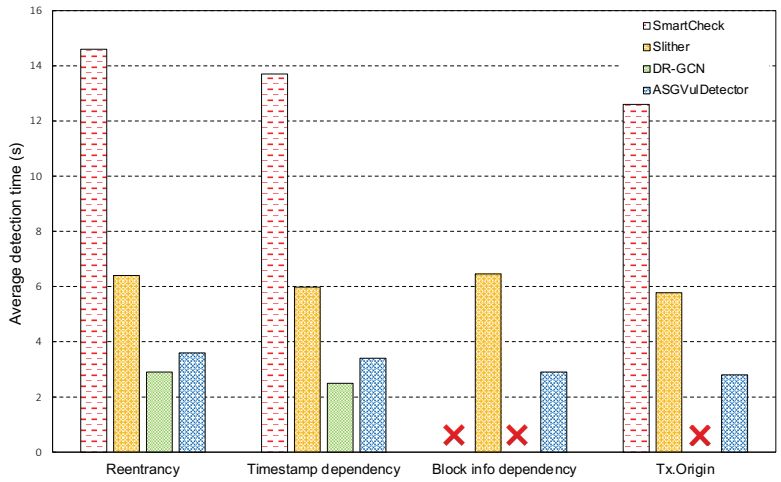


Figure 6. Comparison of detection time for *ASGVulDetector*.

In Figure 6, *SmartCheck* spends more than 10 seconds on vulnerability detection, most of which is consumed by its XPath pattern matching. In contrast, *Slither* has more stable and efficient detection time, i.e., six seconds on average in the four cases. This is contributed by its specific intermediate representation. As two methods based on machine learning technique, *DR-GCN* and *ASGVulDetector* have a low detection time, since most work is accomplished during the model training period. However, the time cost of our approach is slightly higher than *DR-GCN*, which is mainly caused by the extra matching step in GMN. Theoretically, the cost of the GNN embedding model used in *DR-GCN* is in the order of $O(|V| + |E|)$, while the cost of that in GMN is $O(|V| \times |V|)$, where $|V|$ and $|E|$ are the numbers of nodes and edges, respectively. Nevertheless, the average detection time of *ASGVulDetector* is only around three seconds, which can be acceptable in most applications.

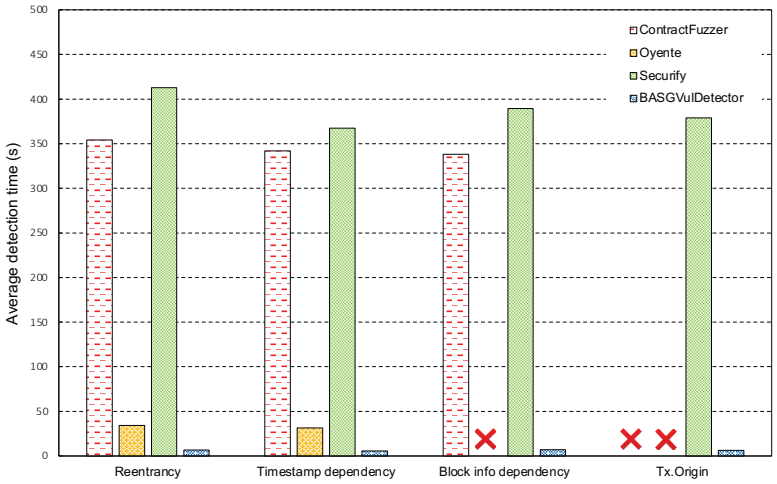


Figure 7. Comparison of detection time for *BASGVulDetector*.

In Figure 7, *ContractFuzzer* takes nearly 350 seconds on average to find a bug since it needs to actually run the EVM code for certain rounds. *Oyente* spends much less time than *ContractFuzzer*, around 33 seconds per each possible vulnerability detection, which is mainly consumed by constraint solvers. *Securify* is regarded as the most time-expensive one among the four methods, as it needs to transform the bytecode to a specific language and traverse all patterns to detect violations. *BASGVulDetector* has the least detection time, only about six seconds on average. However, compared to *ASGVulDetector*, it costs more time since generating and embedding the contract graph for the bytecode is more complex than those for the source code.

5.4. Impact of GMN Parameters

Due to the fact that the performance of deep neural network is usually affected by its hyper-parameters, in this experiment, we investigated the impacts of different parameters on the performance of vulnerability detection. Considering the fact that our two approaches utilize the same training model, we only analyzed *ASGVulDetector* in detail, and gave the experimental results of *BASGVulDetector* straightforwardly. We considered four key parameters, i.e., number of epochs, embedding dimension, number of hidden layers, and learning rate. The proper values obtained in this experiment were applied to performance evaluation as aforementioned. We chose F1 score to reveal the above impacts; the results of *ASGVulDetector* are summarized in Figure 8.

Figure 8a shows the impact of different epoch settings. With the increasing number of epochs, the F1 scores of the four types of vulnerabilities all grow dramatically before the data point of 80. After that, the detection performance enters into a stable state, suggesting that 80 epochs are sufficient to train the neural network. Figure 8b demonstrates the impact of different embedding dimensions. At the beginning of all four cases, F1 scores increase significantly with the embedding dimension. However, as the dimension grows larger than 110, the values of the F1 scores drop sharply. This can be explained by small dimensions limiting the expressiveness of the embedding, but too-large ones suffering from overfitting. Figure 8c shows the impact of different numbers of hidden layers in GMN. Hidden layers are of key importance to capture the internal features of a contract graph, but a large number of hidden layers will significantly increase the complexity of the model. This rule can be derived from this experiment, where four hidden layers are enough to extract internal connections inside contract graphs. Figure 8d reveals the impact of different learning rates. The learning rate is another key hyper-parameter of the neural network, which helps with network convergence and thus influences detection performance. In this experiment, we started from a large learning rate, i.e., 0.1, and scaled the value down to find an appropriate rate, which is suggested to be 0.001.

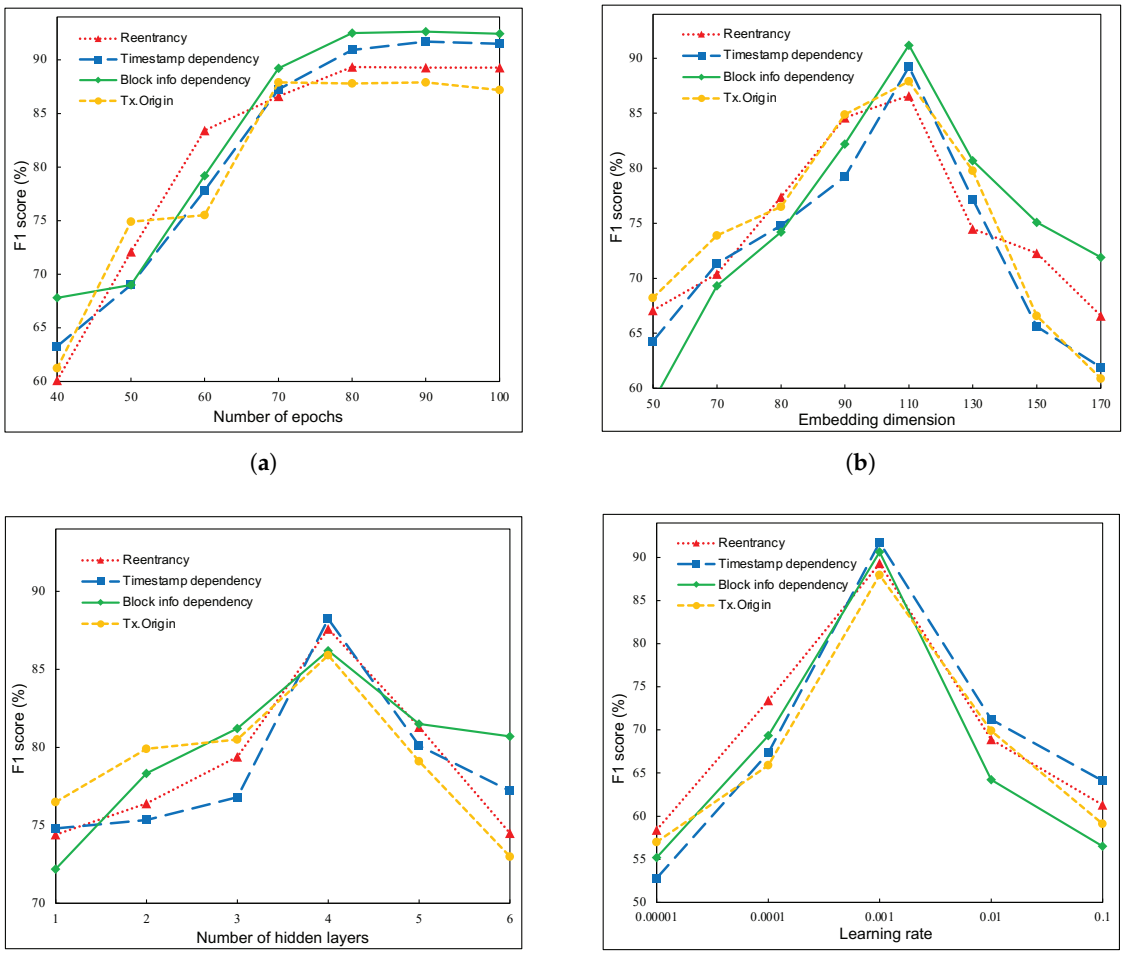


Figure 8. Impact of GMN key parameters in *ASGVulDetector*. (**a**) Impact of epochs. (**b**) Impact of embedding dimension. (**c**) Impact of number of hidden layers. (**d**) Impact of learning rate.

In *BASGVulDetector*, recommended values for the four parameters, i.e., number of epochs, embedding dimension, number of hidden layers, and learning rate, are 70, 90, 4, and 0.001, respectively. It is worth noting that *BASGVulDetector* requires fewer epochs and embedding dimensions than *ASGVulDetector* since the contract graph in *BASGVulDetector* is less complicated than that in *ASGVulDetector*.

5.5. Impact of Different Models

Theoretically, GMN improves GNN by introducing a cross-graph matching mechanism. In the last experiment, we investigated the superiority of GMN to GNN with respect to vulnerability detection in smart contracts. We utilized GNN to perform the same processes as GMN did, and adopted the same parameters where applicable. Figures 9 and 10 illustrate the performance impacts of different models for *ASGVulDetector* and *BASGVulDetector*, respectively. In all cases, GMN outperforms GGNN. On average, GMN improves GGNN on F1 score by roughly 25% in *ASGVulDetector* and around 19% in *BASGVulDetector*, respectively.

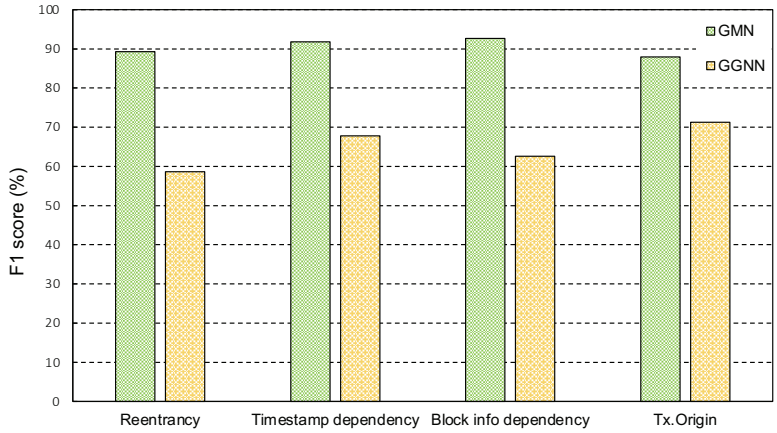


Figure 9. Impact of different models for *ASGVulDetector*.

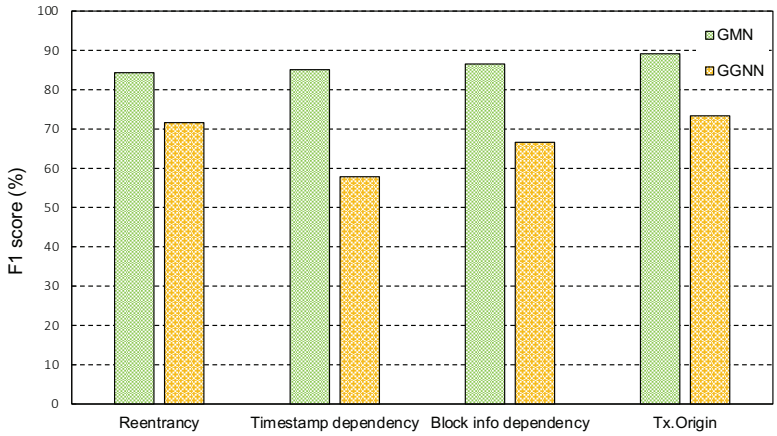


Figure 10. Impact of different models for *BASGVulDetector*.

6. Conclusions and Future Work

Vulnerability detection is crucial for blockchain-based smart contracts. Existing studies have provided a number of promising methods through different software-testing techniques, i.e., fuzzing, symbolic execution, formal verification as well as static analysis. In this paper, we proposed two static analysis tools *ASGVulDetector* and *BASGVulDetector* for smart contracts in source code and bytecode, respectively. We designed a novel graph representation for smart contracts and utilized the pioneer graph-based deep learning technique GMN for vulnerability detection. The contract graph is able to reflect more syntactic and semantic information, while GMN provides a good performance guarantee of similarity learning. Experimental results reveal that *ASGVulDetector* improves the best of three source-code only static analysis tools (i.e., *SmartCheck*, *Sliter*, and *DR-GCN*) on F1 score by 12.6% on average, while *BASGVulDetector* improves that of the three detection tools supporting bytecode (i.e., *ContractFuzzer*, *Oyente*, and *Securify*) on F1 score by 25.6% on average. We also investigated the parameters of GMN as well as its superiority to the GNN model.

However, there are still limitations to the proposed approaches, which can be summarized as two aspects. On one hand, we consider four types of vulnerabilities in this work, but there exist various types of smart contract vulnerabilities, which contain different syntactic and semantic characteristics [29,30]. On the other hand, the proposed approaches require sampling contracts of different vulnerabilities to train specific models, thus being limited in unknown vulnerability detection. In the future, we plan to study more types of smart contract vulnerabilities to enhance the scalability of the proposed approaches, and develop new methods to detect unknown vulnerabilities. Generalizing the proposed method

to other software testing scenarios, such as vulnerability detection for other programming languages and code clone detection, is another direction for future research.

Author Contributions: The research for this paper was undertaken by Y.Z. and D.L.; conceptualization and investigation, Y.Z. and D.L.; software and validation, D.L.; writing-original draft preparation, Y.Z. and D.L.; writing-review and editing, Y.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded by the Natural Science Foundation of Jiangsu Province of China (Grant No. BK20190346) and the 2019 Industrial Internet Innovation and Development Project, Ministry of Industry and Information Technology, China (Grant No. 6709010003).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data supporting this article are from previously reported studies and datasets, which have been cited.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ethereum. Ethereum: Blockchain App Platform. Available online: https://www.ethereum.org/ (accessed on 13 April 2022).
2. Nick, S. Formalizing and Securing Relationships on Public Networks. *First Monday* **1997**, *2*, 1–21.
3. Khan, S.N.; Loukil, F.; Ghedira-Guegan, C.; Benkhelifa, E.; Bani-Hani, A. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Netw. Appl.* **2021**, *14*, 2901–2905. [CrossRef] [PubMed]
4. Vacca, A.; Sorbo, A.D.; A.Visaggio, C.; Canfora, G. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *J. Syst. Softw.* **2021**, *174*, 110891. [CrossRef]
5. Izhar, M.M.; Louis, S.C.; Alana, G.; Elgar, G.; Gabrielle, F.; Ryan, S.; M, K.H.; Marek, L. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *J. Cases Inf. Technol.* **2019**, *21*, 19–32.
6. Destefanis, G.; Marchesi, M.; Ortu, M.; Tonelli, R.; Bracciali, A.; Hierons, R. Smart contracts vulnerabilities: A call for blockchain software engineering? In Proceedings of the 2018 International Workshop on Blockchain Oriented Software Engineering, Campobasso, Italy, 20 March 2018; pp. 19–25.
7. Enmei, L.; Wenjun, L. Static analysis of integer overflow of smart contracts in ethereum. In Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy, Nanjing, China, 10–12 January 2020; pp. 110–115.
8. Liu, Z.; Qian, P.; Wang, X.; Zhu, L.; He, Q.; Ji, S. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In Proceedings of the 30th International Joint Conference on Artificial Intelligence, Montreal, QC, Canada, 19–26 August 2021; pp. 2751–2759.
9. Zhou, L.; Qin, K.; Cully, A.; Livshits, B.; Gervais, A. On the Just-In-Time Discovery of Profit-Generating Transactions in DeFi Protocols. In Proceedings of the 2021 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 24–27 May 2021; pp. 919–936.
10. Perez, D.; Livshits, B. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In Proceedings of the 30th USENIX Security Symposium, Vancouver, BC, Canada, 11–13 August 2021; pp. 1325–1341.
11. Jiang, B.; Liu, Y.; Chan, W.K. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 259–269.
12. Grieco, G.; Song, W.; Cygan, A.; Feist, J.; Groce, A. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; pp. 557–560.
13. Wüstholz, V.; Christakis, M. Harvey: A Greybox Fuzzer for Smart Contracts. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, 8–13 November 2020; pp. 1398–1409.
14. Nguyen, T.D.; Pham, L.H.; Sun, J.; Lin, Y.; Minh, Q.T. sFuzz: An efficient adaptive fuzzer for solidity smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Korea, 27 June–19 July 2020; pp. 778–788.
15. Luu, L.; Chu, D.H.; Olickel, H.; Saxena, P.; Hobor, A. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 254–269.
16. 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, San Juan, PR, USA, 3–7 December 2018; pp. 664–676.
17. Mossberg, M.; Manzano, F.; Hennenfent, E.; Groce, A.; Grieco, G.; Feist, J.; Brunson, T.; Dinaburg, A. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering, San Diego, CA, USA, 11–15 November 2019; pp. 1186–1189.

18. So, S.; Hong, S.; Oh, H. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In Proceedings of the 30th USENIX Security Symposium, Vancouver, BC, Canada, 11–13 August 2021; pp. 1361–1378.
19. Lin, S.W.; Tolmach, P.; Liu, Y.; Li, Y. SolSEE: A Source-Level Symbolic Execution Engine for Solidity. In Proceedings of the 2022 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–18 November 2022; pp. 1–6.
20. Bai, X.; Cheng, Z.; Duan, Z.; Hu, K. Formal Modeling and Verification of Smart Contracts. In Proceedings of the 2018 7th International Conference on Software and Computer Applications, Kuantan, Malaysia, 8–10 February 2018; pp. 322–326.
21. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Bünzli, F.; Vechev, M. Securify: Practical Security Analysis of Smart Contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 67–82.
22. Albert, E.; Correas, J.; Gordillo, P.; Román-Díez, G.; Rubio, A. SAFEVM: A safety verifier for Ethereum smart contracts. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 386–389.
23. Antonino, P.; Roscoe, A.W. Solidifier: Bounded model checking solidity using lazy contract deployment and precise memory modelling. In Proceedings of the 36th Annual ACM Symposium on Applied Computing, Virtual Event, 22–26 March 2021; pp. 1788–1797.
24. Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. SmartCheck: Static Analysis of Ethereum Smart Contracts. In Proceedings of the 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, Gothenburg, Sweden, 27 May 2018; pp. 9–16.
25. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, Montreal, QC, Canada, 27 May 2019; pp. 8–15.
26. Xue, Y.; Ma, M.; Lin, Y.; Sui, Y.; Ye, J.; Peng, T. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, VIC, Australia, 21–25 September 2020; pp. 1029–1040.
27. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart Contract Vulnerability Detection Using Graph Neural Networks. In Proceedings of the 29th International Joint Conference on Artificial Intelligence, Yokohama, Japan, 11–17 July 2020; pp. 3283–3290.
28. Alharby, M.; Aldweesh, A.; van Moorsel, A. Blockchain-based Smart Contracts: A Systematic Mapping Study. In Proceedings of the 2018 International Conference on Cloud Computing, Big Data and Blockchain, Fuzhou, China, 15–17 November 2018; pp. 1–6.
29. Khan, Z.A.; Namin, A.S. Ethereum Smart Contracts: Vulnerabilities and their Classifications. In Proceedings of the 2020 IEEE International Conference on Big Data, Atlanta, GA, USA, 10–13 December 2020; pp. 1–10.
30. Chen, J.; Xia, X.; Lo, D.; Grundy, J.; Luo, X.; Chen, T. Defining Smart Contract Defects on Ethereum. *IEEE Trans. Softw. Eng.* **2022**, *48*, 327–345. [CrossRef]
31. Fan, W.; Ma, Y.; Li, Q.; He, Y.; Zhao, E.; Tang, J.; Yin, D. Graph Neural Networks for Social Recommendation. In Proceedings of the 2019 World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 417–426.
32. Zhao, L.; Li, Z.; Al-Dubai, A.Y.; Min, G.; Li, J.; Hawbani, A.; Zomaya, A.Y. A Novel Prediction-Based Temporal Graph Routing Algorithm for Software-Defined Vehicular Networks. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 13275–13290. [CrossRef]
33. Li, Y.; Gu, C.; Dullien, T.; Vinyals, O.; Kohli, P. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 3835–3845.
34. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. In Proceedings of the 2018 International conference on learning representations, Vancouver, BC, Canada, 30 April–3 May 2018; pp. 1–17.
35. Contro, F.; Crosara, M.; Ceccato, M.; Preda, M.D. EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode. In Proceedings of the 29th International Conference on Program Comprehension, Madrid, Spain, 20–21 May 2021; pp. 127–137.
36. Wood, G. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Available online: https://ethereum.github.io/yellowpaper/ (accessed on 21 April 2022).
37. Ferreira, J.F.; Cruz, P.; Durieux, T.; Abreu, R. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual Event, 21–25 December 2020; pp. 1349–1352.
38. Ghaleb, A.; Pattabiraman, K. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, 18–22 July 2020; pp. 415–427.