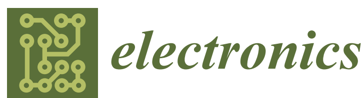

MDPI

Article

Software Weakness Detection in Solidity Smart Contracts Using Control and Data Flow Analysis: A Novel Approach with Graph Neural Networks

Aria Seo [1], Young-Tak Kim [2], Ji Seok Yang [3], YangSun Lee [4,*] and Yunsik Son [1,*]

[1] Department of Computer Science and Engineering, Dongguk University, Seoul 04620, Republic of Korea; seoaria@dgu.ac.kr
[2] Department of Biomedical Sciences, Korea University College of Medicine, Seoul 02841, Republic of Korea; kyt3426@korea.ac.kr
[3] Department of Artificial Intelligence, Dongguk University, Seoul 04620, Republic of Korea; jiseok4404@dgu.ac.kr
[4] Department of Electronics and Computer Engineering, Seokyeong University, Seoul 02713, Republic of Korea
* Correspondence: yslee@skuniv.ac.kr (Y.L.); sonbug@dongguk.edu (Y.S.)

Abstract: Smart contracts on blockchain platforms are susceptible to security issues that can lead to significant financial losses. This study converts the Solidity code into abstract syntax trees and generates control flow graphs and data flow graphs. These graphs train a graph convolutional network model to detect security weaknesses. The proposed system outperforms traditional tools, achieving higher accuracy, recall, precision, and F1 scores when detecting weaknesses such as integer overflow/underflow, reentrancy, delegate call to the untrusted callee, and time-based issues. This study demonstrates that leveraging control and data flow analysis with graph neural networks significantly enhances smart contract security and provides a robust and reliable solution.

Keywords: smart contract security; graph neural network; control flow graph; data flow graph; Solidity weakness detection

Citation: Seo, A.; Kim, Y.-T.; Yang, J.S.; Lee, Y.; Son, Y. Software Weakness Detection in Solidity Smart Contracts Using Control and Data Flow Analysis: A Novel Approach with Graph Neural Networks. *Electronics* **2024**, *13*, 3162. https://doi.org/10.3390/electronics13163162

Academic Editors: Yeong-Seok Seo and Jun-Ho Huh

Received: 3 July 2024
Revised: 5 August 2024
Accepted: 6 August 2024
Published: 10 August 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Recent advancements in blockchain technology, which is known for its decentralized and tamper-resistant characteristics, have led to rapid developments in various applications. Blockchain is essentially a distributed and shared ledger of transactions maintained across a network of miners following a consensus protocol [1]. Smart contracts, which are programs that automatically execute or enforce the terms of a contract, leverage the tamper resistance of blockchain technology to enhance the transparency and reliability of transactions. However, poorly designed smart contracts are susceptible to weaknesses that can be exploited by malicious actors [2]. A notable example is the Decentralized Autonomous Organization (DAO) event, where hackers exploited a reentrance bug in the DAO contract, leading to a theft of 3.6 million Ether [3]. These weaknesses are continuously being discovered and exploited.

Smart contracts on blockchain platforms are susceptible to security issues that can lead to significant financial losses. Existing security analysis tools for smart contracts are primarily rule-based, which makes them specialized for detecting specific weaknesses but prone to generating many false positives. To address these limitations, this study proposes a novel method that considers a program's control and data flows to detect security weaknesses. Our approach begins by converting the Solidity code into an abstract syntax tree (AST) and then generating control flow graphs (CFGs) and data flow graphs (DFGs). These graphs are then used to train a graph convolutional network (GCN) model to classify security weaknesses in the code. The key novelty of our approach lies in the integration of control and data flow analysis with graph neural networks (GNNs),

which provides more comprehensive and accurate detection of weaknesses compared to traditional rule-based tools.

Blockchain networks adopt existing test and dynamic execution methods from programming language communities to detect smart contract weaknesses [4,5]. However, these approaches encounter two significant challenges. First, they rely heavily on predefined rules (or patterns) established by experts, which can result in high false-positive rates and difficulties handling complex code patterns. However, some attackers can easily circumvent these rules [6]. Second, the scalability of these rule-based approaches is inherently limited because they depend on a small group of experts to define accurate rules [7]. With the rapid increase in smart contracts, it has become impractical for a few experts to verify all contracts [8].

To address these limitations and to align them with intelligent approaches for solving software problems using AI techniques, this study proposes a method that utilizes a CFG and a DFG to detect security weaknesses in smart contracts. The CFG represents the execution flow of a program, whereas the DFG represents data flow. By analyzing these graphs, we can effectively detect security weaknesses in complex smart contracts and reduce the number of false positives. This approach is aligned with the fundamental concepts of artificial intelligence, software engineering, and data security.

The proposed method employs GNN to process graph-shaped data. By learning and analyzing graph data using a GNN, we aim to develop a more effective weakness detection tool than existing rule-based tools, thereby enhancing the security of smart contracts and protecting user assets. This leverages the advancements in AI service development, model optimization, and data integration [9].

This study analyzed security weaknesses in Solidity, the core programming language of the Ethereum blockchain platform. Specifically, we targeted four types of weaknesses: reentry, timestamp, overflow/underflow, and delegate call. These weaknesses were selected to demonstrate the effectiveness of the proposed method over existing rule-based detection tools.

We parsed the Solidity code into an AST to consider both the execution and data flows of the program. The AST is represented as a tree structure in JSON format, containing structural elements, execution flow, and data flow information. However, ASTs include extraneous information beyond what is necessary for GNN learning. Therefore, we extracted the CFG and DFG from the AST, focusing on the relevant parts representing the execution and data flows. The CFG nodes represent the basic blocks, the edges represent the control flow, the DFG nodes represent the data processing operations, and the edges represent the data flow.

By integrating these graphs and their attributes into a single dataset, along with the corresponding labels, we generated multiple datasets to train the GNN model. This process allows us to input the Solidity code, convert it into an AST, generate graphs, and determine whether the code contains security weaknesses. Our proposed method aims to provide an effective weakness detection tool that considers both execution and data flow, thereby improving the security of smart contracts.

2. Related Work

2.1. Blockchain and Smart Contracts

Smart contracts are blockchain-based programs that automatically enforce or execute contractual terms when conditions are met [10]. They operate without a centralized intermediary, thus enhancing cost efficiency and reliability over centralized systems. The entire code and execution results of smart contracts are stored in the blockchain, ensuring transparency and trust among the participants. The overall operation of a smart contract involves users developing the contract, which is then converted into an Ethereum Virtual Machine (EVM) bytecode executed by an EVM. This bytecode is stored in the Ethereum network as part of the blockchain. Figure 1 illustrates the general operation of a smart contract.

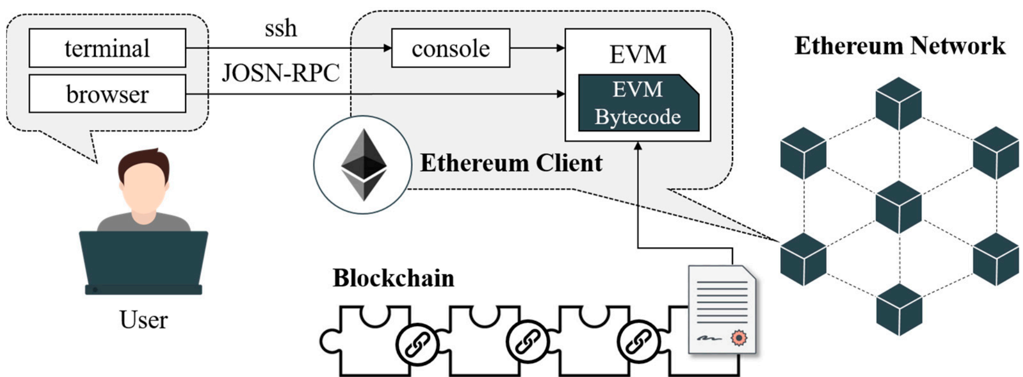

Figure 1. Operation process of a smart contract.

2.1.1. Ethereum and Solidity

Ethereum is a blockchain platform that implements a decentralized Web 3.0 [11]. Similar to Bitcoin, Ethereum is based on blockchain technology, a decentralized ledger maintained by miners following a consensus protocol. Ethereum focuses on the development and execution of smart contracts and decentralized applications (DApps).

The most significant feature of Ethereum includes smart contracts, which are self-executing contracts written in code that are automatically executed when predefined conditions are satisfied. For instance, after purchasing goods, payments are automatically processed once the conditions are fulfilled. Ethereum also employs blockchain to store and verify transactions and contract information, ensuring data integrity without a central authority. Ethereum uses Ether, a cryptocurrency, as a value exchange to execute smart contracts or process transactions.

Ethereum uses a gas unit to execute smart contracts or process transactions in the network. Gas represents the computational resources required for processing, which limits the resources required to prevent network overloading or malicious contract execution.

Ethereum is structured into four primary layers: the application, consensus, data, and network layers [12]. The application layer develops and executes DApps based on smart contracts, enabling user interactions through browsers or applications without a central intermediary. The consensus layer comprises consensus protocols that validate transactions and generate blocks, initially using proof of work (PoW) but transitioning to proof of stake (PoS) to address issues such as high energy consumption and mining centralization. The data layer stores and manages various data types in the blockchain, including smart contracts, transactions, and state data. Finally, the network layer represents the infrastructure that manages the Ethereum blockchain, involving nodes and P2P communications to ensure network stability and security.

Solidity is a high-level programming language designed to develop smart contracts on the Ethereum blockchain [13]. It allows the writing and deployment of smart contracts executed on an EVM. Solidity's syntax is similar to that of C++, JavaScript, and Python, incorporating familiar elements such as declarations, data types, access modifiers, and functions. It enables the development of trustless and automated applications by allowing code to interact with the blockchain directly. This interaction includes reading and writing data and managing assets (typically Ether) without a central intermediary. Moreover, it supports complex data types and control structures, enabling developers to create intricate smart contracts capable of performing various functions, ranging from simple transactions to more complex operations involving multiple parties. It also includes features that enhance security and prevent common weaknesses. For example, it supports custom error handling, function modifiers controlling access to contract functions, and events facilitating monitoring logging and contract interactions. These features help developers write more secure and reliable smart contracts, which are crucial for maintaining the integrity and trustworthiness of blockchain applications.

2.1.2. Smart Contract Weakness Classification

The smart contract weakness classification (SWC) framework standardizes categorizing patterns and weaknesses related to smart contracts. Inspired by common weakness enumeration (CWE), SWC is tailored to smart contracts' unique environments and conditions. Each weakness in the SWC has a unique ID and includes descriptions, severity levels, attack scenarios, preventive measures, and relevant code examples.

Smart contract weaknesses can have severe consequences, including financial losses and a loss of trust in blockchain platforms. Notable weaknesses include reentrancy, where an attacker can repeatedly call a function before the initial execution is completed, and integer overflow/underflow, resulting in unexpected behaviors due to arithmetic errors. Other common weaknesses include timestamp dependence, where contract execution relies on the block timestamp, and improper delegation calls, which can expose the contract to unauthorized access.

The SWC framework aims to assist developers in identifying and mitigating weaknesses by providing a comprehensive and structured approach to smart contract security. By adhering to the guidelines and best practices outlined in SWC, developers can improve the security and robustness of their smart contracts, reduce the risk of exploitation, and enhance overall trust in the blockchain ecosystem. Table 1 presents the metrics provided by SWC, including unique identifiers (IDs), weakness titles, and their corresponding CWE IDs (relationships).

Table 1. SWC IDs and weakness titles.

SWC ID	Title	Related CWE ID
SWC-100	Function default visibility	CWE-710
SWC-101	Integer overflow and underflow	CWE-682
SWC-102	Outdated compiler version	CWE-937
SWC-103	Floating pragma	CWE-664
SWC-104	Unchecked call return value	CWE-252
SWC-105	Unprotected Ether withdrawal	CWE-284
SWC-106	Unprotected SELFDESTRUCT instruction	CWE-284
SWC-107	Reentrancy	CWE-841
SWC-108	State variable default visibility	CWE-710
SWC-119	Uninitialized storage pointer	CWE-824
SWC-110	Assert Violation	CWE-670
SWC-111	Use of deprecated solidity functions	CWE-477
SWC-112	Delegatecall to untrusted callee	CWE-829
SWC-113	DoS with failed call	CWE-703
SWC-114	Transaction order dependence	CWE-362
SWC-115	Authorization through tx. origin	CWE-477
SWC-116	Block values as a proxy for time	CWE-829

These weaknesses highlight the security concerns developers must address when writing smart contracts. By understanding and implementing measures to mitigate these weaknesses, developers can enhance the security of their applications, thereby contributing to the overall robustness and reliability of the Ethereum ecosystem.

2.2. Program Analysis

Program analysis involves examining and understanding the structure and behavior of code to detect potential issues and optimize its performance. Program analysis is a crucial aspect of software engineering that examines code to understand its behavior, structure, and properties. This process helps improve code quality, optimize performance, and ensure security. This section discusses the main components of the program analysis used in this study: the AST, CFG, and DFG.

2.2.1. Abstract Syntax Tree

An AST is a tree representation of the abstract syntactic structure of the source code [14]. Each node in the AST denotes a construct occurring in the source code, such as variable declarations, operators, or function calls. The AST is generated through parsing, which analyzes the source code to produce a structured representation.

When compiling or interpreting source code, the initial step typically involves converting the code into an AST. This tree provides a clear understanding of the code's syntax and semantics, enabling the identification of the code's flow and data structure. Static code analysis tools utilize ASTs to analyze code structure and detect errors or security weaknesses. In addition, development environments use ASTs to facilitate features such as code refactoring and automatic code generation.

2.2.2. Control Flow Graph and Data Flow Graph

The AST is a foundational element for generating CFGs, essential for detecting security weaknesses in smart contracts.

The CFG represents the execution flow of the code. Each node in the CFG represents a basic block, a straight-line code sequence with no branches except at the entry and exit points. The edges between nodes indicate possible control flow paths [15]. The nodes in the CFG depict entry and exit points within the code, such as function definitions, control statements, and variable declarations. Edges represent conditional statements' true/false branches or loops' entry/exit points. The CFG provides a detailed view of how the program executes, highlighting the various paths the execution may take under different conditions.

Similarly, the DFG illustrates how the data moves within the code. Each node in the DFG represents a data processing operation, and the edges represent the data flow between these operations. Nodes include variable assignments, operations, function calls, and literals, whereas edges indicate how data are transferred from one node to another [16]. DFGs help understand data flow through a program and identify how data are manipulated and passed between different parts of the code. This is crucial for detecting weaknesses related to data handling, such as incorrect variable initialization, unintended data leaks, and improper handling of sensitive information.

These graphs were labeled CFG or DFG and integrated into a single dataset with the corresponding label data. Multiple datasets were generated and fed into the GNN for training. This method enables the analysis and detection of security weaknesses in the Solidity code by considering control and data flows. The proposed approach provides an effective tool for detecting weaknesses in smart contracts by analyzing the execution and data flows within the code.

The program analysis techniques discussed herein, particularly the use of ASTs, CFGs, and DFGs, are integral to our approach to detecting security weaknesses in smart contracts. By leveraging these techniques, we can generate detailed representations of the smart contract's execution and data flows, which are crucial for constructing the graphs required for our GNN model. These graphs provide input data for our machine learning model, enabling the application of advanced AI techniques to identify potential security weaknesses in the code.

In summary, the AST, CFG, and DFG program analysis techniques played vital roles in our methodology. They allow us to systematically deconstruct the smart contract code, extract relevant features, and build the necessary models to enhance the security detection mechanisms. The integration of program analysis with advanced AI techniques underpins the effectiveness of the proposed solution for securing smart contracts on the Ethereum platform.

2.3. Machine Learning for Security Analysis

Machine learning has significantly advanced the field of security analysis by providing sophisticated tools for identifying and mitigating security weaknesses. Traditional security analysis methods often rely on predefined rules and patterns, which are limited in scope

and scalability. In contrast, machine-learning models can learn from large datasets, adapt to new threats, and identify complex patterns that may not be apparent through rule-based approaches.

In smart contracts, machine learning can analyze extensive code datasets, detect anomalies, and identify potential security weaknesses. By training models on known weaknesses, these techniques can detect similar patterns in new, unseen smart contracts, thereby enhancing the robustness and reliability of the security analysis.

Incorporating machine learning into security analysis has numerous advantages. Machine learning models can handle large volumes of data and scale efficiently, making them well-suited for environments with substantial data inputs. They are adaptable and respond to new and evolving threats more flexibly than static rule-based systems. Moreover, machine learning excels in recognizing complex patterns and correlations within data, which is crucial for identifying subtle weaknesses that may otherwise go unnoticed.

2.3.1. Comparative Analysis of Machine Learning Models for Smart Contract Security

Several machine learning (ML) models, each with strengths and weaknesses, can be employed for smart contract security analysis. Convolutional neural networks (CNNs) are highly effective for image and grid-like data analysis, excelling in detecting spatial hierarchies. In smart contract security, CNNs can potentially analyze the visual representations of code or detect patterns in structured data. However, their fixed grid structure limits their ability to process graph-structured data, which is essential for understanding the complex interdependencies within smart contracts.

Recurrent neural networks (RNNs) and long short-term memory (LSTM) networks are designed for sequential data processing, making them suitable for tasks involving temporal dependencies such as code sequences or execution traces. They can model the sequence of operations within a smart contract, helping to identify the patterns that lead to weaknesses. However, these studies struggle to capture the complex relational structures crucial for smart contract analysis.

Random forest and gradient boosting machine (GBM) are ensemble learning methods known for their robustness and effectiveness in many classification tasks. Smart contract security can classify code snippets or detect anomalies based on predefined features. However, these models cannot naturally incorporate the relational information in graphs, limiting their effectiveness in capturing intricate dependencies in smart contract codes.

GNNs are designed to operate on graph-structured data. Smart contracts can be naturally represented as graphs, where nodes represent code elements (e.g., functions, variables) and edges represent relationships (e.g., data flows, control flows). GNNs leverage these structures to capture local and global dependencies, making them particularly well-suited for detecting complex security weaknesses.

GNNs offer several key advantages in smart contract security analyses. They capture the complex relationships and dependencies within the code, which are crucial for identifying security issues that depend on control and data flows. GNNs are highly scalable and can efficiently process large graphs, making them suitable for analyzing increasingly complex smart contracts deployed on blockchain platforms. Finally, GNNs exhibit strong generalization capabilities, allowing them to generalize from training data to new, unseen data and improve the detection of novel weaknesses not explicitly represented in the training dataset. This generalization is essential for maintaining robust security as new weaknesses emerge and smart contract technology evolves.

By leveraging the unique strengths of GNNs, the proposed method provides a comprehensive and accurate tool for detecting security weaknesses in smart contracts. This approach considers the execution and data flows within the code. It integrates advanced AI techniques to enhance security detection mechanisms, offering a robust solution for improving the reliability and safety of smart contracts.

2.3.2. Graph Neural Networks

GNNs represent a powerful class of deep learning models designed to operate on graph-structured data. Unlike traditional neural networks, which process data in grid-like structures such as images or sequences, GNNs can handle graphs' intricate, non-Euclidean structures. This capability makes GNNs particularly suitable for tasks involving relationships and interactions between entities, such as analyzing smart contracts.

Smart contract codes can naturally be represented as graphs, with CFGs and DFGs capturing the flow of execution and data. GNNs can effectively analyze and detect security weaknesses by converting smart contracts into graph representations.

GNNs offer several advantages for this application. First, they capture the complex relationships and dependencies within the code that are crucial for identifying security issues. They can efficiently process large graphs, making them suitable for analyzing complex smart contracts. In addition, GNNs can generalize training data to new, unseen data, thereby improving the detection of novel weaknesses.

The basic operation of a GNN model involves several steps, as illustrated in Figure 2. Initially, each node in the graph is assigned a feature vector representing various node attributes. During the message-passing phase, nodes gather information from their neighbors. This information is aggregated using the mean, sum, or max functions. After passing messages, the nodes update their feature vectors using neural network layers incorporating aggregated information. A readout function is then applied to extract the final graph representation, which can be used for various tasks such as node or graph classification [17].

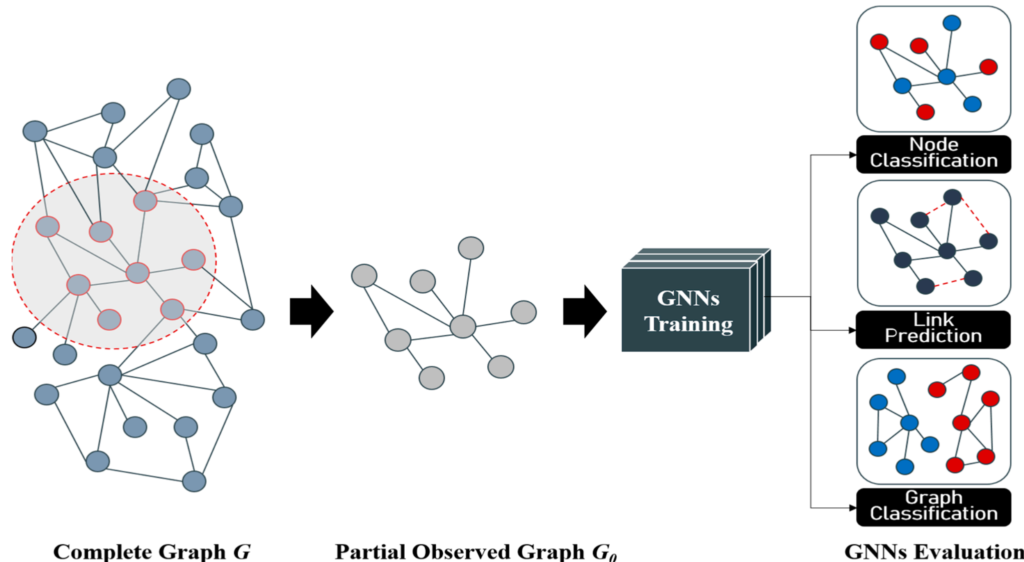

Figure 2. GNN model architecture.

In this study, we leverage GNNs to analyze graph representations of smart contracts. By converting the contract code into a CFG and a DFG, we can utilize GNNs' powerful capabilities to capture complex patterns and relationships within the code. This approach enables more effective detection of a wide range of security weaknesses than traditional methods.

The integration of GNNs into the proposed methodology provides several benefits. GNNs enable the detection of subtle and complex weaknesses that can be overlooked using rule-based detection tools. They efficiently process and analyze large smart contracts, making them suitable for real-world applications. Furthermore, GNNs are robust to code variations, thereby improving the reliability of weakness detection.

By employing GNNs, we aimed to develop a comprehensive and effective tool for smart contract security analysis. This approach contributes to blockchain applications' overall security and reliability by offering a robust mechanism for detecting and mitigating

potential security weaknesses in smart contracts. Specifically, we employed a basic GNN model, the Graph Convolutional Network (GCN), to analyze the generated CFGs and DFGs. The GCN model is trained to classify the types of security weaknesses present in the code, thereby enhancing the security detection capabilities of smart contracts.

3. Proposed System

The proposed system is a software weakness analysis tool for smart contracts written in Solidity, a high-level programming language used to develop smart contracts on the Ethereum blockchain. This system converts the Solidity code into an AST representing the syntactic structure of the program and then generates CFGs and DFGs from the AST. The CFGs and DFGs were subsequently used to train a GNN model to detect security weaknesses in the code. This multistep approach allows the system to analyze the execution and data flows within smart contracts effectively. Figure 3 shows the overall architecture of the proposed system.

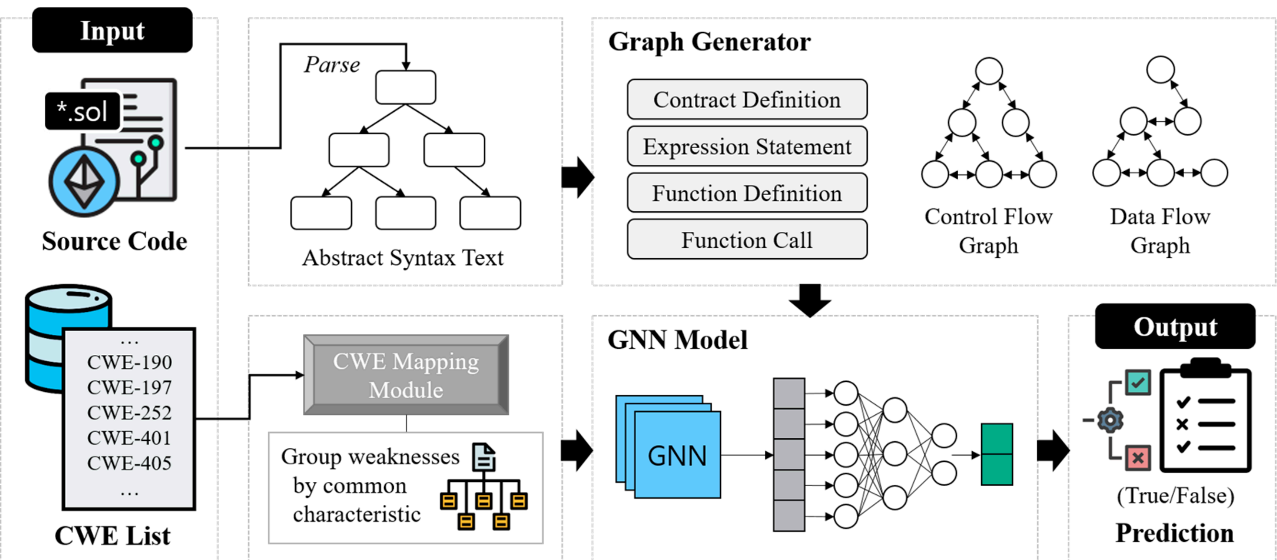

Figure 3. Overall architecture of the proposed system.

Algorithm 1 illustrates the pseudocode of the proposed system. Initially, the Solidity code is translated into an AST using the **Translate_AST_from_Solidity** function. From this AST, the **Generate_CFG_from_AST** function creates a list of CFGs. An empty container, **viz_code**, is initialized to store the dot-format visualization code. For each CFG in the list, the CFG is converted to dot format using **Convert_CFG_to_dot** and appended to **viz_code**. The complete visualization code in **viz_code** is then transformed into a DGL dataset format using **Convert_viz_to_DGL**. A dataset object is created with the **Create_Dataset** function, including the DGL dataset and corresponding labels. This dataset is then batched using a data loader initialized by **Create_DataLoader**, which shuffles the data for robust training. Next, the GNN model is instantiated with **Initialize_GNN**, and an optimizer is initialized using **Initialize_Optimizer** with the model parameters. The model is trained over a predefined number of epochs. For each batch of data, the model performs a forward pass with **Model_Forward_Pass**, computes the loss with **Compute_Loss**, and updates the model parameters using the optimizer with **Update_Model**. Finally, the trained model is saved using the **Save_Model** function, resulting in a classification model ready for deployment.

Algorithm 1 Pseudocode for the process of the proposed system

```
Input:   Solidity code file
Output:  Trained classification model
   1:    ast = Translate_AST_from_Solidity(file)
   2:    cfg_list = Generate_CFG_from_AST(ast)
   3:    viz_code = ""
   4:
   5:    for cfg in cfg_list do
   6:    viz_code += Convert_CFG_to_dot(cfg)
   7:
   8:    dgl_dataset = Convert_viz_to_DGL(viz_code)
   9:    dataset = Create_Dataset(dgl_dataset, labels)
  10:    dataloader = Create_DataLoader(dataset, batch_size, shuffle=True)
  11:    model = Initialize_GNN()
  12:    optimizer = Initialize_Optimizer(model_parameters)
  13:
  14:    for epoch in range(num_epochs) do
  15:    for batched_graph, labels in dataloader do
  16:    logits = Model_Forward_Pass(model, batched_graph)
  17:    loss = Compute_Loss(logits, labels)
  18:    Update_Model(optimizer, loss)
  19:
  20:    Save_Model(model)
```

This comprehensive process ensures that the system can effectively learn from the graph representations of smart contracts, thereby allowing accurate detection of security weaknesses. Using CFGs and DFGs to represent different aspects of code behavior enables a GNN to capture complex relationships and dependencies within the code, leading to improved performance over traditional rule-based methods.

3.1. AST Generation from Solidity Code

The first step in implementing the proposed system was to convert the Solidity code into an AST using a Solidity parser. Converting the Solidity code to an AST involves several stages: tokenization, syntax parsing, AST generation, and AST analysis and processing. The selected security weaknesses are classified according to the SWC registry, which provides a standardized method for identifying and categorizing smart contract weaknesses [18].

In the tokenization stage, the Solidity parser reads the Solidity code and breaks it into tokens. These tokens represent the smallest elements of the code, such as keywords, variable names, operators, and literature. Each token provides information about the structure and content of the code.

The parser analyzes the tokenized code during syntax parsing to create an AST. This involves checking the code's structure and grammar and generating a tree structure that reflects the code's hierarchical organization. Each node in the AST represents a syntactic construct, such as function definitions, control statements, and variable declarations. The resulting tree structure is then converted into an AST, which provides an abstract representation of the code structure.

Through this process, the Solidity code is transformed into an AST. The AST is typically represented in the JSON format and contains essential information about the execution flow and data flow within the code. This structured representation enables the extraction of CFGs and DFGs from the AST, which are crucial for further analysis.

By converting the Solidity code into an AST, we obtained a comprehensive representation that captured both the syntactic and semantic details of the code. This AST is the foundation for generating the CFGs and DFGs used to train the GNN model for security analysis.

3.2. Generation of Control Flow Graph and Data Flow Graph

The AST represents the overall execution flow and data flow of the code. However, the direct use of the AST for GNN training includes unnecessary information and is not in graph form, making it unsuitable for GNNs. Thus, we extract information representing only the control flow from the AST to generate a CFG and data flow information to generate a DFG.

The steps in generating a CFG include initialization, AST traversal, basic block creation, control structure handling, node connections, and final processing. In the initialization stage, the start and end nodes of the CFG are created, and the current node pointer is set as the start node. During the AST traversal, the AST is traversed in pre-order, processing each node. Continuous statements form a basic block, and new basic blocks begin with control structures, such as conditionals and loops. Control structures, such as conditionals (e.g., if-else) and loops (e.g., for, while), are processed by adding the corresponding CFG nodes and connecting them appropriately to represent true/false branches and loop structures. Jump statements (e.g., break, continue, and return) are handled by adding CFG nodes and making the necessary connections, such as connecting return statements directly to the end node. After processing all AST nodes, the final CFG is formed by connecting the last CFG node to the end node.

To generate a DFG, we analyzed the program's data flow and tracked the relationships between variable definitions and uses. This process involves AST traversal, tracking variable definitions and uses, handling operators and expressions, processing function calls and returns, considering the control flow, and optimizing the DFG by removing unused variables or dead codes. This generates a DFG from the AST, representing the data dependencies within the code.

Algorithm 2 and Figure 4 illustrate the CFG and DFG generated from the Solidity code example. These visualizations aid in understanding how the control and data flows are represented in graph form for further analysis.

Algorithm 2. Example of a CFG generated from Solidity code

```
 1: ...
 2: contract BREBuy {
 3:     struct ContractParam {
 4:         uint32 totalSize;
 5:         uint256 singlePrice;
 6:         uint8 pumpRate;
 7:         bool hasChange;
 8:     }
 9:     uint32 gameIndex = 0;
10:     ContractParam public setConfig;
11:     ContractParam public curConfig;
12:     address[] public addressArray = new address[](0);
13:     function startNewGame() private {
14:         gameIndex++;
15:         if(curConfig.hasChange) {
16:             if(curConfig.totalSize != setConfig.totalSize) {
17:                 curConfig.totalSize = setConfig.totalSize;
18:             }
19:             if(curConfig.singlePrice != setConfig.singlePrice){
20:                 curConfig.singlePrice = setConfig.singlePrice;
20:             }
21:             if(curConfig.pumpRate != setConfig.pumpRate) {
22:                 curConfig.pumpRate = setConfig.pumpRate;
23:             }
24:             curConfig.hasChange = false;
25:         }
26:         addressArray.length=0;
27: }
28: ...
```

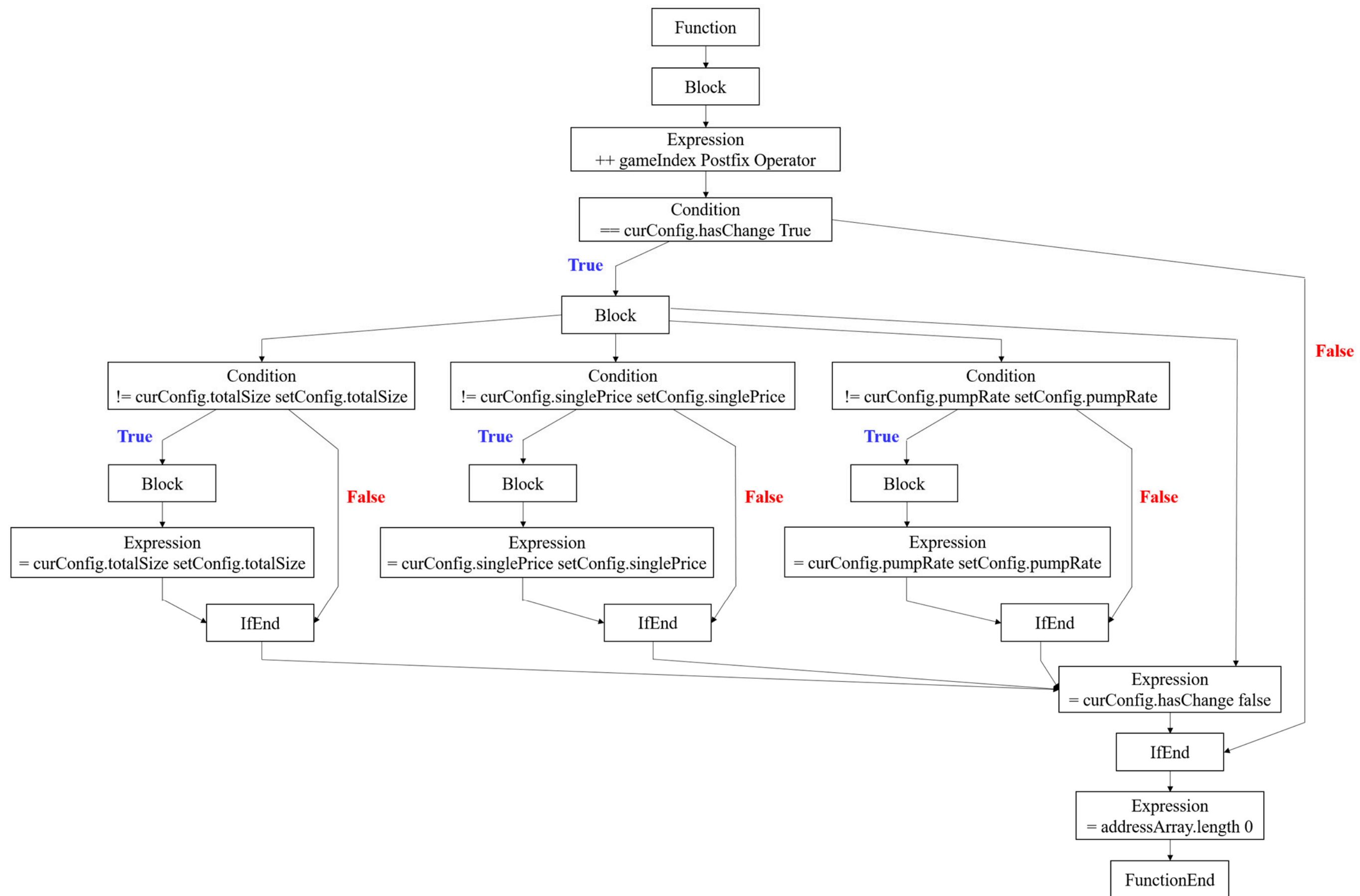

Figure 4. Example of function-level CFG generation in a Solidity contract.

Algorithm 2 shows the CFG generated using the Solidity code. This algorithm shows how the execution flow is captured, with the nodes representing the basic blocks and the edges representing the flow between them. Figure 4 illustrates the CFG generated for each function within the Solidity contract. This example shows how multiple functions within a single contract can be visualized, highlighting the control flow for each function.

By processing the AST into CFGs and DFGs, we can capture both the execution and data flow within the code, enabling effective analysis and weakness detection using GNNs.

3.3. Training the GNN Model

After converting the Solidity code into an AST and generating the corresponding CFG and DFG, these graphs were prepared to train the GNN model. The GNN model used in our study consists of three layers, including two convolutional layers followed by one fully connected layer. Each layer utilizes the ReLU activation function to introduce non-linearity. To prevent overfitting, we applied dropout with a rate of 0.5. The model was trained using the Adam optimizer with a learning rate of 0.001. We conducted the training over 100 epochs with a batch size of 32. The training process involved early stopping criteria to monitor the validation loss and stop training when the validation loss did not improve for 10 consecutive epochs. The generated CFG and DFG contain node and edge information essential for creating graph representations suitable for GNN training.

The node and edge information are stored in two separate files: **nodes.csv** for node information and **edges.csv** for edge information. Each file includes the identifiers and attributes necessary to reconstruct the graphs. Specifically, the nodes.csv file contains columns for **graph_id**, **num_node**, and **type**. The **graph_id** uniquely identifies each graph, **num_node** indicates the number of nodes, and **type** distinguishes between CFG (0) and DFG (1). The edges.csv file includes columns for **graph_id**, **src**, **dst**, and **type**, with **src** and

dst specifying each edge's start and end nodes. These CSV files served as the input data for the GNN model training process, with nodes and edges encoded to facilitate learning.

Each graph can be uniquely identified and reconstructed using the 'graph_id' present in both nodes.csv and edges.csv. These CSV files were then used to generate the final graph data structures required for the GNN training. This is accomplished using a DGL, which helps create and manipulate graph data for deep learning applications.

Figure 5 depicts the conversion of a CFG into a visual representation using Viz, facilitating an understanding of the code control flow. The visualization was then converted into a format suitable for deep learning (DGL), enabling the use of graph data to train the GNN. The graph structure includes the number of nodes and edges, node types, edge types, and metagraphs, which provide an overview of the relationships between the different node types.

```
Number of nodes: 514190
Number of edges: 34173
Node types: ['Block', 'BooleanLiteral', 'Condition', 'Expression', ...]
Edge types: ['normal', 'normal', 'normal', 'normal', 'normal', 'normal',
'normal', 'normal', 'normal', 'false', 'false', 'false', 'false', 'false',
'false', 'false', 'true', 'true', 'true', 'true', 'true', 'normal', 'nor-
mal', ...]
Metagraph: [('Block', 'Condition'), ('Block', 'Expression'), ('Block',
'FunctionEnd'), ('Block', 'IfEnd'), ('Block', 'LoopVariable'), ('Block',
'break'), ('Block', 'return'), ('Block', 'throw'), ('Condition', 'Block'),
('Condition', 'Block'), ('Condition', 'Condition'), ('Condition',
'ForEnd'), ('Condition', 'IfEnd'), ('Condition', 'IfEnd'), ('Condition',
'NumberLiteral'), ...]
```

Figure 5. Example of training graph structure.

Using this graph structure, a program's complex execution and data flow can be effectively represented and learned. The GNN model leverages this representation to analyze and detect potential security weaknesses within the smart contract code. By capturing the intricate relationships and dependencies within the code, the GNN model provides a robust mechanism for identifying subtle and complex security issues that traditional static analysis tools may miss.

4. Experiments and Results

4.1. Selection and Description of Security Weaknesses

To validate the proposed system, we selected security weaknesses from the SWC list, in which existing detection tools are not fully detected. The selection criteria included the severity of the weakness, if it occurred, the inability of existing detection tools to reliably identify the weakness, and the feasibility of detecting the weakness by analyzing the execution and data flows of the program. Based on these criteria, we identified four security weaknesses: SWC-101 (integer overflow and underflow), SWC-107 (reentrancy), SWC-112 (delegatecall to untrusted callee), and SWC-116 (Block values as a proxy for time). The selected weaknesses are summarized in Table 2.

Table 2. Selected security weaknesses.

SWC ID	Title	Related CWE ID	Content
SWC-101	Integer overflow and underflow	CWE-682	Incorrect calculation
SWC-107	Reentrancy	CWE-841	Improper enforcement of behavioral workflow
SWC-112	Delegatecall to untrusted callee	CWE-829	Inclusion of functionality from untrusted control sphere
SWC-116	Block values as a proxy for time	CWE-829	Inclusion of functionality from untrusted control sphere

- **SWC-101 (integer overflow and underflow)**: Integer overflow and underflow weaknesses occur when an integer variable exceeds its maximum or falls below its minimum limit. Integer overflow occurs when an integer variable exceeds its maximum value, which can cause unexpected initialization to zero or overwrite the other variables. Conversely, integer underflow occurs when an integer variable falls below its minimum value, potentially causing unexpected initialization to the maximum value or overwriting other variables. These weaknesses are critical in the Solidity code, as they can lead to significant economic losses if Ether values are tampered with in smart contracts. An example of this weakness is shown in Algorithm 3.
- **SWC-107 (reentrancy)**: Reentrancy weaknesses occur when a smart contract calls an external contract or function and the external contract or function calls back to the original contract before the first invocation is complete. This can result in unexpected states or drain the balance of contracts. Algorithm 4 shows an example of this weakness.
- **SWC-112 (delegatecall to untrusted callee)**: The delegatecall weakness occurs when the delegatecall function in Solidity executes code from another contract within the context of the caller's contract. If not handled properly, this can lead to security issues, because the contract can manipulate the state of the calling contract. An example of this weakness is shown in Algorithm 5.
- **SWC-116 (block values as a proxy for time)** arises when Solidity smart contracts use block values such as block.timestamp or now as a proxy for time. Miners can manipulate such dependencies, leading to unexpected behaviors or invalid contract states. Algorithm 6 shows an example of this weakness.

Algorithm 3 Example of Solidity code with integer overflow and underflow weakness

```
1: ...
2: contract IntegerOverflowMappingSym {
3:     mapping(uint256 => uint256) map;
4:     function init(uint256 k, uint256 v) public{
5:         map[k] -= v;
6:     }
7: }
8: ...
```

Algorithm 4 Example of Solidity code with reentrancy weakness

```
1: ...
2: contract SimpleDAO {
3:     mapping(address => uint) public credit;
4:     function donate(address to) payable public {
5:         credit[to] += msg.value;
6:     }
7:     function withdraw(uint amount) public {
8:         if (credit[msg.sender] >= amount) {
9:             require(msg.sender.call.value(amount)());
10:            credit[msg.sender] -= amount;
11:        }
12:    }
13:    function queryCredit(address to) view public returns (uint) {
14:        return credit[to];
15:    }
16: }
17: ...
```

Algorithm 5 Example of Solidity code with delegatecall weakness

```
1:  ...
2:  contract Proxy {
3:          address owner;
4:          constructor() public {
5:                owner = msg.sender;
6:          }
7:          function forward(address callee, bytes _data) public {
8:                require(callee.delegatecall(_data));
9:                }
10: }
11: ...
```

Algorithm 6 Example of Solidity code with block values as a proxy for time weakness

```
1:  ...
2:  contract TimeBasedVault {
3:          address owner;
4:          uint256 unlockTime;
5:          constructor(uint256 _unlockHours) public {
6:                owner = msg.sender;
7:                unlockTime = now + (_unlockHours * 1 hours);
8:          }
9:          function withdraw() public {
10:               require(msg.sender == owner, "Only the owner can withdraw.");
11:               require(now >= unlockTime, "Funds are locked until the unlock time.");
12:               msg.sender.transfer(address(this).balance);
13:                                            }
14: }
15: ...
```

To evaluate the performance of the proposed system, we used a dataset collected from the Ethereum platform and GitHub repositories, which included instances of selected weaknesses. The dataset consisted of 372 instances of SWC-101, 382 instances of SWC-107, 202 instances of SWC-112, and 504 instances of SWC-116. Each weakness is categorized into safe codes or codes that contain weaknesses. Table 3 summarizes the numbers of safe and at-risk instances for each weakness type.

Table 3. Number of data instances per software weakness.

Weakness	SWE-101		SWE-107		SWE-112		SWE-116	
Label	0	1	0	1	0	1	0	1
Count	278	94	209	173	127	75	278	226

The selected weaknesses and their respective datasets were used to train and validate the proposed system, demonstrating its effectiveness in detecting security issues in solid, smart contracts.

4.2. Selection of Comparative Analysis Tools

Experiments were conducted using existing analytical tools on the constructed dataset to demonstrate the validity and significance of the proposed system. The selection criteria for these tools included being open source, freely available, and commonly used for analyzing security weaknesses in Solidity smart contracts. Based on these criteria, we selected four software weakness analysis tools for comparative analysis: sFuzz, Smartcheck, Osiris, and Mythril. The characteristics of each tool are as follows.

sFuzz [19] is an automated tool for detecting weaknesses in smart contracts. It analyzes Solidity-based smart contracts to identify weaknesses. sFuzz automatically examines smart contracts for various weaknesses and addresses common security concerns. It can detect various weakness types, including reentry, integer overflow/underflow, and

timestamp issues. In addition, sFuzz can generate automated transactions to invoke contract functions and analyze transaction results. The tool produces summary reports that include details regarding the discovered weaknesses, their locations in the smart contract, and the outcomes of the executed transactions.

Smartcheck [20] is another tool for detecting security weaknesses in Solidity-based smart contracts. It analyzes the Solidity code to identify security issues, such as reentrancy, integer overflow/underflow, logic errors, and incorrect permission settings. A key feature of SmartCheck is its web-based interface, which allows users to upload smart contracts and review the results easily. The tool automatically generates a report summarizing the findings, including information regarding the identified weaknesses, their locations in the code, detection methods, and risk levels.

Osiris [21] is also designed to detect smart contracts' security weaknesses. It focuses on weaknesses such as reentry, integer overflow/underflow, logic errors, and incorrect permission settings. Similar to SmartCheck, Osiris provides a web interface for users to visualize results. A notable feature of Osiris is its application programming interface (API), which enables integration with other systems, making it easy to incorporate into development and automated security testing processes.

Mythril [22] is a tool for analyzing the security weaknesses of Ethereum smart contracts. It targets the codes of smart contracts running on the Ethereum blockchain platform, detects various security issues, and provides relevant information. Mythril performed both static and dynamic analyses of smart contracts. Static analysis allows code to be examined without executing it, thus identifying weaknesses early in development. On the other hand, dynamic analysis allows Mythril to execute the smart contract and monitor for potential weaknesses and issues during execution.

By comparing the performance of these selected tools against the proposed system on the constructed dataset, we can evaluate the effectiveness of the proposed approach in detecting security weaknesses in Solidity smart contracts.

4.3. Training and Evaluation of the Proposed Model

To validate the effectiveness of the proposed system, we utilized a dataset from Jiang [4] that included samples of Solidity smart contract codes with specific security weaknesses. The dataset focuses on four weaknesses: SWC-101 (integer overflow and underflow), SWC-107 (reentrancy), SWC-112 (delegatecall to untrusted callee), and SWC-116 (block values as a proxy for time). Each weakness category contained labeled samples indicating whether the code was unexposed (label 0) or exposed (label 1), with the following distribution: 372 samples for both unexposed and exposed SWC-101; 382 samples for both unexposed and exposed SWC-107; 202 samples for both unexposed and exposed SWC-112; and 504 samples for both unexposed and exposed SWC-116. This balanced dataset facilitates the robust training and evaluation of the proposed security analysis model.

Four key metrics were used to evaluate the performance of the proposed model and the existing tools: accuracy (ACC), recall (RE), precision (PRE), and F1-score (F1). These metrics are essential for evaluating the models' performance and providing a comprehensive view of their effectiveness in detecting security weaknesses.

Figure 6 illustrates the performance of different analysis tools across the four evaluated security weaknesses using radar charts. These charts provide a visual comparison of the tools' performance based on the four key metrics: accuracy (ACC), recall (RE), precision (PRE), and F1-score (F1). Each subfigure focuses on a specific weakness: (a) SWE-101, (b) SWE-107, (c) SWE-112, and (d) SWE-116. The radar charts demonstrate that the proposed GNN-based model consistently outperforms existing tools across all metrics, showcasing its robustness and reliability in detecting security weaknesses in Solidity smart contracts.

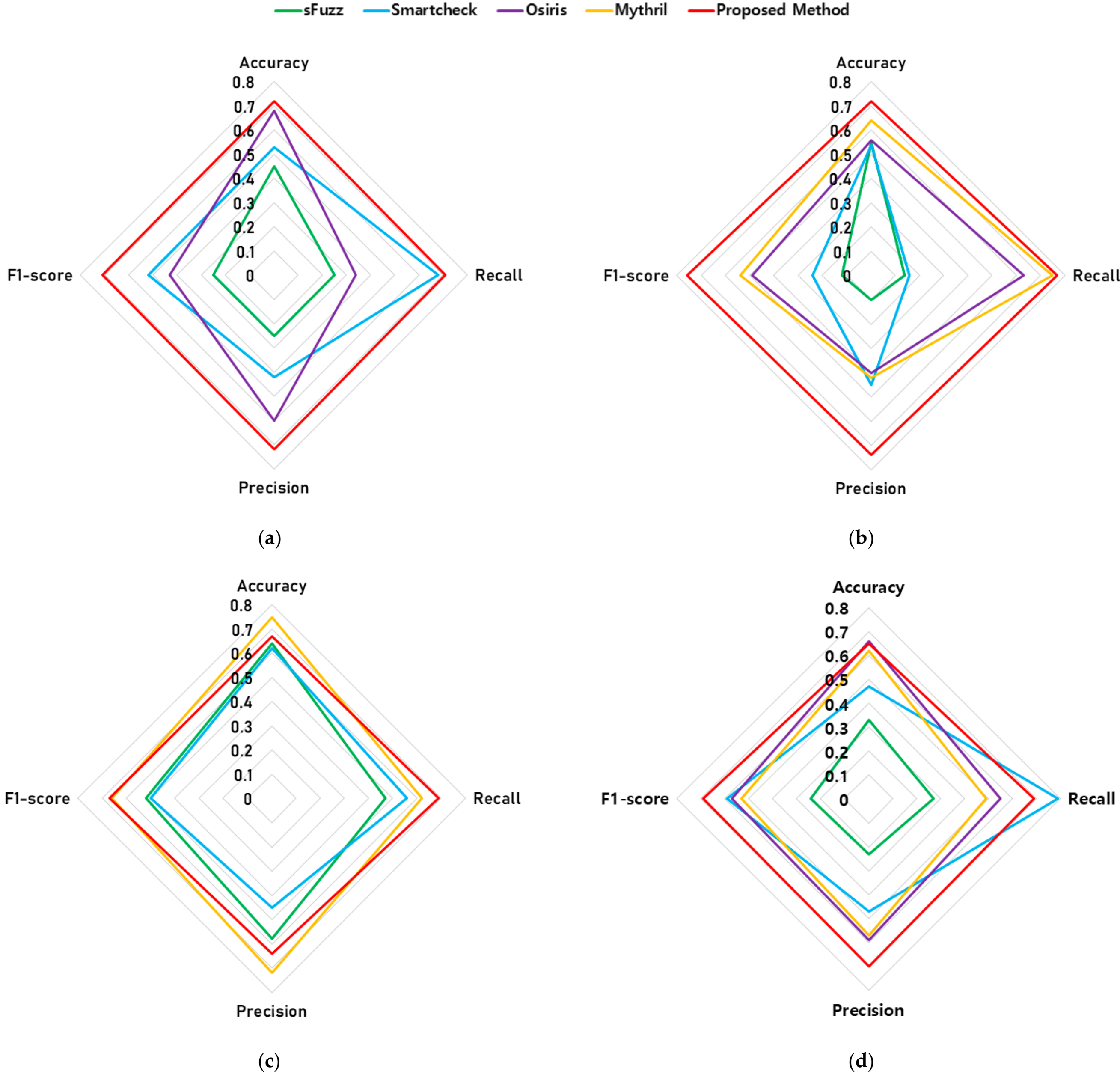

Figure 6. Radar charts illustrating the performance of different analysis tools across various metrics for each security weakness: (**a**) SWE-101, (**b**) SWE-107, (**c**) SWE-112, and (**d**) SWE-116.

Table 4 presents the results of training the proposed model and the comparative experiments with existing tools. This table summarizes the performance of each tool across the four selected security weaknesses.

The results show that the proposed GNN-based model achieves significant performance improvements across all evaluated security weaknesses compared with existing tools. The proposed model's accuracy, recall, precision, and F1-scores were consistently high, demonstrating its effectiveness in detecting security issues in Solidity smart contracts. In contrast, existing tools exhibit varying performance levels, excelling at detecting specific weaknesses and failing in others.

Table 4. Experimental results of different analysis tools in terms of accuracy (ACC), recall (RE), precision (PRE), and F1-score (F1).

Weakness/Tools	SWE-101				SWE-107				SWE-112				SWE-116			
	ACC	RE	PER	F1	ACC	RE	PER	F1	ACC	RE	PER	F1	ACC	RE	PER	F1
sFuzz [19]	0.45	0.25	0.25	0.25	0.55	0.14	0.10	0.12	0.64	0.47	0.58	0.52	0.33	0.27	0.23	0.24
Smartcheck [20]	0.53	0.68	0.42	0.52	0.54	0.16	0.45	0.24	0.62	0.56	0.45	0.50	0.47	0.79	0.47	0.59
Osiris [21]	0.68	0.34	0.60	0.43	0.56	0.63	0.40	0.49	n/a	n/a	n/a	n/a	0.66	0.55	0.59	0.57
Mythril [22]	n/a	n/a	n/a	n/a	0.64	0.75	0.42	0.54	0.75	0.62	0.72	0.66	0.62	0.49	0.57	0.53
Proposed Method	**0.72**	**0.71**	**0.72**	**0.71**	**0.72**	**0.77**	**0.74**	**0.76**	**0.67**	**0.69**	**0.64**	**0.67**	**0.65**	**0.69**	**0.70**	**0.69**

This comprehensive evaluation highlights the robustness and efficacy of the proposed model, suggesting that it provides a reliable tool for smart contract security analysis by capturing intricate relationships and dependencies within the code more effectively than traditional static analysis tools.

5. Conclusions

In the current landscape, most security analysis tools for smart contracts are rule-based, making them specialized for detecting specific weaknesses and prone to generating many false positives. To address these limitations, this study proposes a novel method that integrates control and data flow analysis with GNNs to detect security weaknesses in Solidity smart contracts. This approach begins by converting the Solidity code into an AST, then generating a CFG and a DFG, which are used to train a GCN model to classify security weaknesses.

Our method offers a significant advancement over traditional rule-based tools by providing a robust and reliable solution for detecting a wide range of weaknesses. The experimental results demonstrate that our system outperforms existing rule-based detection tools, achieving higher accuracy, recall, precision, and F1 scores in identifying weaknesses such as integer overflow/underflow, reentrancy, delegate calls to untrusted callees, and time-based issues. These findings highlight the practical benefits of combining control and data flow analysis with GNNs to enhance smart contract security.

The key contribution of this research lies in the innovative integration of program analysis techniques with advanced AI models. By leveraging CFGs and DFGs, our system captures intricate relationships within the code, often overlooked by traditional tools, leading to more comprehensive and accurate detection of weaknesses.

While our method shows significant improvements, there are limitations and challenges to consider. One limitation is the computational complexity associated with generating and analyzing CFGs and DFGs for large and complex smart contracts. The GNN model requires substantial computational resources for training, which may not be readily available in all settings. The accuracy of the model also depends on the quality and diversity of the training dataset, making the availability of comprehensive datasets a potential challenge. Future research should focus on addressing these limitations and optimizing the computational efficiency of our method.

The experimental results demonstrate the effectiveness and significance of the proposed system. As more datasets are collected and training is conducted on additional weaknesses, the system is anticipated to detect a broader range of security weaknesses. Techniques such as generating synthetic datasets have shown promise in enhancing model training and evaluation. This study utilized a basic GCN model, but higher performance is expected by using more advanced and specialized GNN models tailored to specific datasets.

This study also suggests future research directions, such as improving efficiency by simplifying graphs to include only parts related to security weaknesses and exploring more advanced GNN models tailored to specific datasets. Additionally, our method can be extended to incorporate privacy-preserving techniques, such as those used in Blockshare and SymmeProof, to enhance both security and privacy in smart contracts. Combining our approach with other blockchain security methods could further enhance overall security, providing a more comprehensive and resilient solution.

In summary, this study highlights the potential for integrating control flow and data flow analysis with advanced AI techniques to enhance the detection of security weaknesses in smart contracts. Similar to using advanced computational techniques in medical analyses for precise diagnostics, the proposed approach uses AI to improve software weakness detection. The findings suggest that focusing on the relevant sections of code for graph generation and exploring more sophisticated GNN models can further improve the effectiveness and efficiency of the system.

This study's contributions are significant in enhancing the detection accuracy of smart contract weaknesses and addressing the limitations of existing tools by reducing false positives and covering a wider range of weaknesses. This study paves the way for future advancements by promoting safer and more reliable blockchain environments.

Author Contributions: Conceptualization, A.S., Y.-T.K., J.S.Y., Y.L. and Y.S.; Data curation, Y.-T.K. and J.S.Y.; Formal analysis, A.S., Y.L. and Y.S.; Project administration, Y.L. and Y.S.; Software, A.S., Y.-T.K. and J.S.Y.; Supervision, Y.L. and Y.S.; Validation, A.S. and Y.L.; Visualization, A.S., Y.-T.K., J.S.Y. and Y.S.; Writing—original draft, A.S., Y.-T.K. and J.S.Y.; Writing—review & editing, A.S., Y.L. and Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data used in this study can be accessed from the following sources (accessed on 5 August 2024): EEA EthTrust Security Levels Specification version 2: [https://entethalliance.org/ethtrust-security-levels-specification-v2/] (https://entethalliance.org/ethtrust-security-levels-specification-v2/)—GitHub, SmartContractSecurity/SWC-registry: Smart Contract Weakness Classification and Test Cases: [https://github.com/SmartContractSecurity/SWC-registry] (https://github.com/SmartContractSecurity/SWC-registry).

Acknowledgments: The research was supported by a National Research Foundation of Korea (NRF) grant from the Korean government (MSIT) (No. 2018R1A5A7023490). This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-2020-0-01789), and the Artificial Intelligence Convergence Innovation Human Resources Development (IITP-2024-RS-2024-00254592) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation). This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT (No. 2022R1F1A1063340)).

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

References

1. Zheng, Z.; Xie, S.; Dai, H.N.; Chen, X.; Wang, H. Blockchain challenges and opportunities: A survey. *Int. J. Web Grid Serv.* **2018**, *14*, 352–375. [CrossRef]
2. Nguyen, D.H.; Seo, A.; Nnamdi, N.P.; Son, Y. False Alarm Reduction Method for Weakness Static Analysis Using BERT Model. *Appl. Sci.* **2023**, *13*, 3502. [CrossRef]
3. Tsankov, P.; Dan, A.; Drachsler-Cohen, D.; Gervais, A.; Buenzli, 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.
4. 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.
5. Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; Roscoe, B. Reguard: Finding reentrancy bugs in smart contracts. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, Gothenburg, Sweden, 27 May–3 June 2018; pp. 65–68.

6. He, D.; Deng, Z.; Zhang, Y.; Chan, S.; Cheng, Y.; Guizani, N. Smart contract vulnerability analysis and security audit. *IEEE Netw.* **2020**, *34*, 276–282. [CrossRef]
7. Yu, H.; Nnamdi, N.P.; Seo, A.; Park, J.; Son, Y. Motility Analysis of Diaphragm in Patients with Chronic Pulmonary Lung Disease Based on Computed Tomography Technique. *IEEE Access* **2023**, *11*, 101544–101555. [CrossRef]
8. Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; He, Q. Smart contract vulnerability detection using graph neural networks. In Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, Yokohama, Japan, 7–15 January 2021; pp. 3283–3290.
9. Alabdulwahab, S.; Kim, Y.T.; Seo, A.; Son, Y. Generating Synthetic Dataset for ML-Based IDS Using CTGAN and Feature Selection to Protect Smart IoT Environments. *Appl. Sci.* **2023**, *13*, 10951. [CrossRef]
10. Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.B.D.; Xia, X.; Feng, Y.; Xu, B. Smart contract development: Challenges and opportunities. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2084–2106. [CrossRef]
11. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
12. Kushwaha, S.S.; Joshi, S.; Singh, D.; Kaur, M.; Lee, H.N. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access* **2022**, *10*, 6605–6621. [CrossRef]
13. Wohrer, M.; Zdun, U. Smart contracts: Security patterns in the ethereum ecosystem and solidity. In Proceedings of the 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), Campobasso, Italy, 20 March 2018; pp. 2–8.
14. Baxter, I.D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone detection using abstract syntax trees. In Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, 20 November 1998; No. 98CB36272, pp. 368–377.
15. Allen, F.E. Control flow analysis. *ACM Sigplan Not.* **1990**, *5*, 1–19. [CrossRef]
16. Allen, F.E.; Cocke, J. A program data flow analysis procedure. *Commun. ACM* **1976**, *19*, 137. [CrossRef]
17. Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The graph neural network model. *IEEE Trans. Neural Netw.* **2008**, *20*, 61–80. [CrossRef] [PubMed]
18. Zheng, Z.; Su, J.; Chen, J.; Lo, D.; Zhong, Z.; Ye, M. Dappscan: Building large-scale datasets for smart contract weaknesses in dapp projects. *IEEE Trans. Softw. Eng.* **2004**, *50*, 1360–1373. [CrossRef]
19. 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, Republic of Korea, 5–11 October 2020; pp. 778–788.
20. Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. Smartcheck: Static analysis of ethereum smart contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, Gothenburg, Sweden, 27 May–3 June 2018; pp. 9–16.
21. 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.
22. Mueller, B. A Framework for Bug Hunting on the Ethereum Blockchain 2017. Available online: https://github.com/ConsenSys/mythril (accessed on 5 December 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.