Article

TACSan: Enhancing Vulnerability Detection with Graph Neural Network

Qingyao Zeng [1,†], Dapeng Xiong [2,*,†], Zhongwang Wu [2], Kechang Qian [2], Yu Wang [2] and Yinghao Su [1]

[1] Institute of Graduate, Space Engineering University, Beijing 101416, China; qingyao_zeng@hgd.edu.cn (Q.Z.)
[2] Institute of Aerospace Information, Space Engineering University, Beijing 101416, China
* Correspondence: xiongdapeng@hgd.edu.cn
† These authors contributed equally to this work.

Abstract: With the increasing scale and complexity of software, the advantages of using neural networks for static vulnerability detection are becoming increasingly prominent. Before inputting into a neural network, the source code needs to undergo word embedding, transforming discrete high-dimensional text data into low-dimensional continuous vectors suitable for training in neural networks. However, analysis has revealed that different implementation ideas by code writers for the same functionality can lead to varied code implementation methods. Embedding different code texts into vectors results in distinctions that can reduce the robustness of a model. To address this issue, this paper explores the impact of converting source code into different forms on word embedding and finds that a TAC (Three-Address Code) can significantly eliminate noise caused by different code implementation approaches. Given the excellent capability of a GNN (Graph Neural Network) in handling non-Euclidean space data and complex features, this paper subsequently employs a GNN to learn and classify vulnerabilities by capturing the implicit syntactic structure information in a TAC. Based on this, this paper introduces TACSan, a novel static vulnerability detection system based on a GNN designed to detect vulnerabilities in C/C++ programs. TACSan transforms the preprocessed source code into a TAC representation, adds control and data edges to create a graph structure, and then inputs it into the GNN for training. Comparative testing and evaluation of TACSan against other renowned static analysis tools, such as VulDeePecker and Devign, demonstrate that TACSan's detection capabilities not only exceed those methods but also achieve substantial enhancements in accuracy and F1 score.

input: preprocessed source code into a TAC representation that adds control and data edges to create a graph structure that is then inputted into the GNN

Keywords: static analysis; graph neural network; intermediate representation; vulnerability detection; software security

check for updates

Citation: Zeng, Q.; Xiong, D.; Wu, Z.; Qian, K.; Wang, Y.; Su, Y. TACSan: Enhancing Vulnerability Detection with Graph Neural Network. *Electronics* **2024**, *13*, 3813. https://doi.org/10.3390/electronics13193813

Academic Editor: Hung-Yu Chien

Received: 22 August 2024
Revised: 23 September 2024
Accepted: 24 September 2024
Published: 26 September 2024

1. Introduction

With the rapid advancement of technology, the expansion of system applications is continuous, but so are the security issues that accompany them. Many systems have become vulnerable and susceptible to attacks [1] due to a lack of attention to security. Therefore, effective vulnerability detection is of paramount importance. Vulnerability detection encompasses dynamic analysis, static analysis, and hybrid analysis techniques [2]. Dynamic analysis requires the monitoring and detection of a program's state through its execution. In contrast, static analysis does not require program execution; it identifies vulnerabilities by abstractly modeling the code. Hybrid analysis integrates both static and dynamic approaches. As software upgrades iterate continuously and the complexity of functionalities grows, the prevalence of vulnerabilities has also risen. Static analysis has garnered widespread attention due to its advantages such as minimal resource consumption and rapid detection time.

Static analysis technology can be divided into traditional static analysis and learning-based static analysis technology. Traditional static analysis techniques require abstracting and modeling, followed by the application of artificially preset vulnerability analysis rules, which heavily rely on expert experience. Due to the strong advantages of machine learning in different fields [3,4], experts have tried to introduce deep learning into vulnerability detection, and learning-based static analysis techniques have developed rapidly [5].

Generally speaking, learning-based static analysis techniques can be roughly divided into two stages: source code preprocessing and neural network-based learning. During the preprocessing phase, the source code can be transformed into different forms, such as an AST (Abstract Syntax Tree) and a TAC, which are two IRs (Intermediate Representations) generated during the program compilation process. Based on an AST or a TAC, various graph structures can be generated by adding data edges or control edges. After undergoing word embedding, these diverse graph structures will serve as inputs to the neural network for learning and training. Alternatively, without any processing, the source code, an AST, or a TAC can be directly word-embedded and then fed into the neural network. Similarly, there are three forms of learning and training that can be conducted by the neural network: sequence-based learning, tree-based learning, and graph-based learning. If the input is a sequence of source code or TAC, sequence-based learning is employed; if the input is an AST, tree-based learning is used; and if the input is an AST or TAC with added edge information, graph-based learning can be applied.

However, at the source code level, when implementing the same functionality, the same programming language often has multiple ways of implementation. Different implementation methods can achieve the same functionality, which is easily understandable to humans, but for the models, different implementation methods mean different words are fed in, and the vector representations generated after word embedding will also be different. The differences between vectors imply that noise is introduced due to different implementation methods, reducing the robustness of the model. In response, it is necessary to investigate which form of preprocessing in the source code can minimize the noise impact caused by different code implementation methods. This paper addresses the aforementioned issue and finds that when source code is transformed into the TAC representation generated by the GCC (GNU Compiler Collection), it significantly mitigates the impact brought about by different code implementation methods.

Due to the excellent capability of GNNs in handling non-Euclidean spatial data and complex features, they play a crucial role in the field of software security in terms of neural network selection. For instance, consider a variable that is initialized and declared at the beginning of a code but only used at the end; when the entire source code is directly input into the neural network, the connection between initialization and usage becomes very weak and difficult to detect. However, with a GNN, this relationship can be represented through clear edges, enabling the neural network to effectively learn data flow. Another example is loop structures, which indicate that a program may repeatedly jump between a few lines of code. If the entire source code is merely encoded and input into the neural network, this structural information will be lost. GNNs excel at preserving these relationships, making them very effective in capturing subtleties in source code patterns. To retain the implicit syntactic structure information in a TAC, the generated TAC is chosen to be converted into a graph structure before being input into the Graph Neural Network.

Because of this, this paper introduces TACSan (Three-Address Code Sanitizer), a GNN security analysis tool based on a TAC that mitigates the noise from diverse implementation methods. TACSan first preprocesses the function-level source code by comment removal, naming standardizer, and code translation. Afterward, the generated TAC is augmented with data and control edges. Then, it undergoes graph embedding to become a vector representation, which is fed into the GNN for training and ultimately performs binary classification to determine the presence or absence of vulnerabilities.

In conclusion, the main contributions of this paper can be summarized as follows:

- We propose converting source code to a TAC in the preprocessing phase, which will reduce the problem of model robustness degradation caused by different code implementation methods.

- TACSan, a novel security analysis tool utilizing a TAC and a GNN, automates and streamlines the extraction of vulnerability features, facilitating binary classification of function-level source code.
- Five distinct types of vulnerability datasets were constructed in this study, and the results demonstrate that TACSan significantly enhances the accuracy of vulnerability detection over other static analysis tools.

The rest of this article is organized as follows. First, related work is presented in Section 2, and the relevant theories and motivations are introduced in Section 3. Subsequently, Section 4 details the overall design and process of TACSan and explains the key modules and steps, respectively, including the preprocessing module, the graph module, and the classification module. Comparative evaluation experiments are conducted in Section 5. Section 6 discusses the current shortcomings, and Section 7 presents the final conclusions.

2. Related Work

Learning-based static source code analysis is crucial for software security, and many security researchers have conducted studies in this area. This section discusses related work from three perspectives: sequence-based learning, tree-based learning, and graph-based learning.

2.1. Sequence-Based Learning

The statistical similarities between programming and natural languages [6] make it feasible to represent source code as sequences for vulnerability detection.

Li et al. [7] first identified code snippets related to library/API function calls, forming program slices that were organized into a series of statements interrelated on control or data flow and then fed into a BiLSTM (Bidirectional Long-Short Time Memory neural network) model for training. Xu et al. [8], after identifying code snippets related to library/API function calls, used natural language processing techniques to analyze these snippets, converting them into feature vectors before inputting them into a CLSTM (Contextual LSTM) model for training. Saccente et al. [9] utilized an LSTM (Long Short-Term Memory) model for static function-level vulnerability detection in Java source code. Zou et al. [10] proposed a BiLSTM network aimed at integrating different types of features from code gadgets and code attention. Li et al. [11] extracted code snippets reflecting vulnerability syntactic features SyVCs (Syntax-based Vulnerability Candidates), converted them into vulnerability candidates containing semantic information SeVCs (Semantics-based Vulnerability Candidates), and encoded these candidates into vectors that were fed into deep learning models, such as Bi-GRU (Bidirectional Gated Recurrent Unit), for training and vulnerability detection.

2.2. Tree-Based Learning

However, the simple use of sequences as a representation method fails to capture the structured information inherent in source code. Consequently, most subsequent studies have shifted away from using sequences and have opted for tree or graph-based representations instead [12]. An AST, a tree structure generated during the compilation of source code, is one of the most widely used forms of representation. Dam et al. [13] transformed source code into an AST representation and applied a Tree-LSTM model for software defect detection, validating its effectiveness through evaluations on two datasets. Feng et al. [14] also converted source code into an AST and utilized Bi-GRU along with enhancements for variable-length vectors. Their results indicate that the model achieves higher accuracy and a lower false positive rate compared to other analysis tools. Tian et al. [15] introduced TrVD, which decomposes an AST and employs a tree LSTM network for analyzing source code, and further applies a Transformer encoder to aggregate semantic information from subtrees to identify potential vulnerabilities in target code fragments. The outcomes demonstrate that TrVD has delivered significant advancements in the accuracy, efficiency, and practicality of vulnerability detection.

2.3. Graph-Based Learning

In addition to utilizing an AST solely as neural network input, subsequent research has focused on harnessing various graph structures based on an AST to better capture multidimensional features of source code, such as control and data flow information. Zhou et al. [16] proposed Devign, which integrates the AST, CFG (control flow graph), DFG (data flow graph), and the natural code sequence of each function into a unified graph. This graph is then fed into a gated graph neural network and a convolutional layer for code classification. Meng et al. [17] introduced HybridNN, leveraging an HCG (Hybrid Code graph) derived from a CPG (code property graph) as input for a graph neural network. This approach significantly enhances the performance of software vulnerability detection. Where a CPG consists of an AST, CFG, and PDG (program dependency graph). Abdu et al. [18] presented a novel DH-CNN (deep hierarchical convolutional neural network) model. By combining the AST with multi-source representations of CFG and DDG (data dependency graph), the DH-CNN effectively enhances the accuracy of software defect prediction.

3. Background and Motivation

This section begins with an introduction to the foundational knowledge of the theories involved in this paper. It then uses the for loop as an example to explore the impact of different representations of source code, which plays a key role in the establishment of the TACSan model.

3.1. Graph Neural Network

GNNs are a type of deep learning model specifically designed for processing graph-structured data. Unlike traditional deep learning models that focus on processing regular-structured data such as images or sequences, GNNs are specifically designed for processing irregular and dynamically changing graph-structured data, which have led to significant achievements in many emerging tasks [19,20]. Similarly, numerous security professionals have introduced GNNs into the field of security for research purposes. Therefore, it is necessary to provide a brief introduction to GNNs to better understand their concepts and applications.

In a GNN, each node represents an entity in the graph, and each edge represents a relationship or connection between nodes. This structure enables GNNs to effectively integrate and utilize information about nodes and their neighborhoods, thus supporting complex inference and prediction tasks [21–23].

One of the key advantages of GNNs is their ability to integrate local information and the global structure of nodes, enhancing the representational power through multi-level information transfer and aggregation. The essence of the GNN model lies in the messaging mechanism, where nodes update their representations by passing and aggregating information between neighboring nodes. This iterative information transfer process enables GNNs to gradually extract the feature representations of the nodes within the overall graph, better supporting subsequent tasks such as classification, regression, or graph structure prediction.

GNNs can be categorized into three types based on the level of the task: graph-level, edge-level, and node-level. Graph-level tasks concentrate on the properties and characteristics of the entire graph, typically classifying, generating, or predicting properties at the global level. Applications include detecting properties of entire communities in social networks or making global qualitative predictions about chemical molecules. Node-level tasks, on the other hand, focus on operations and predictions for individual nodes, such as node classification, clustering, and representation learning. Examples include user classification in social networks, item recommendation in recommendation systems, and protein function prediction in bioinformatics. Edge-level tasks involve modeling and predicting the features and attributes of each edge within the graph, such as predicting edge existence, type, or weight. In social networks, this could involve predicting relationships between users or assessing correlations in recommendation systems. In this paper, source code vulnerability detection is approached as a graph-level task. GNNs can effectively model the control and data flows within software, classifying entire functions as vulnerable or secure by inputting them into the GNN.

3.2. Three Address Code

A TAC is an intermediate language representation used for code optimization and code generation in compilers and interpreters. It translates high-level source code into a simpler operation level, which enhances the comprehensibility and processability of the program's control and data flow. The generation of a TAC is accomplished through syntactical and semantic analysis stages, which convert complex source code structures into a simplified three-address form, preparing the groundwork for subsequent optimization and code generation.

In a TAC, each instruction typically consists of one operator and up to three operands. Operators may include arithmetic, logical, or other operations supported by the intermediate code. Operands may encompass variables, constants, or compiler-generated temporary registers or variables. This streamlined structure facilitates the compiler's ability to perform optimizations such as register allocation, dead code elimination, and constant propagation, thereby enhancing the program's efficiency and performance.

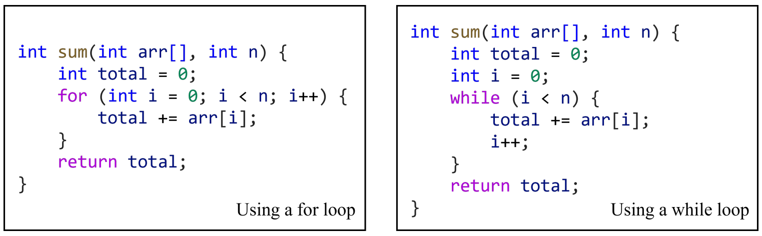
The use of TACs extends beyond the internal optimization processes of compilers to various program analysis tools and IRs of programming languages. By standardizing source code into a unified three-address form, the complexities of conversion and parsing between different programming languages are reduced while preserving the structural and semantic integrity of the code. This unified IR offers a common foundation for software tools to conduct a range of static and dynamic analysis techniques, including static checking, program slicing, and performance analysis. In terms of specific compiler implementations, there are variations in TAC implementations, such as GCC's middle representation is GIMPLE IR, Clang/LLVM's middle representation is LLVM IR. This article converts the source code to a GIMPLE IR output at some stage during the GCC compilation process.

3.3. Motivation

Neural networks, as a powerful machine learning tool, have demonstrated exceptional performance across various domains. They are capable of identifying patterns and predicting outcomes by extracting complex statistical correlations from vast amounts of data, and in some tasks, they have even surpassed human capabilities. For instance, in fields such as image recognition, speech processing, and natural language understanding, deep learning models have achieved remarkable success. However, despite the advantages of neural networks in handling large volumes of data and executing specific tasks, they also have certain limitations due to the fundamental differences in the way they learn and process information compared to humans. Neural networks rely on extracting statistical patterns from large datasets, whereas humans learn by constructing mental concepts and understanding the deep structure of language. This approach to information processing implies that when the form of data to be processed changes, neural networks may struggle to adapt and generalize to different contexts.

In the field of software security, for the implementation of the same functionality, there are often numerous different approaches even within the same programming language. This is easily understood by humans, as human developers can choose different design and coding strategies based on experience, style, or performance requirements. However, for neural networks, this diversity may increase the complexity of identifying and learning security vulnerabilities. Neural networks typically require a large amount of training data to learn patterns, and these data often include multiple different implementation methods, which can introduce noise due to varying code implementations, thereby reducing the robustness of the neural network model.

Therefore, it is necessary to explore which form of source code preprocessing can reduce the noise impact caused by different implementation methods. Here, taking the 'for' loop and 'while' loop in the C language as examples, the aim is to investigate the impact of two IRs, AST and TAC, on the different implementation methods of source code. As shown in Figure 1, there are two different implementation methods for summing the first n items of an array 'arr'. The first method uses a 'for' loop, and the second uses a 'while' loop. It is evident that due to the dissimilarity in the source code sequences of the two methods, different vector representations will be generated during subsequent word embedding, thereby introducing noise and reducing the robustness of the neural network model.



```
int sum(int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += arr[i];
    }
    return total;
}
```

```
int sum(int arr[], int n) {
    int total = 0;
    int i = 0;
    while (i < n) {
        total += arr[i];
        i++;
    }
    return total;
}
```

Figure 1. Representing two different implementation methods using source code.

Figure 2 is a visual representation of the ASTs corresponding to the two implementation methods generated by Joern, an open-source analysis platform based on CPGs. In this paper, the different nodes of the two ASTs are highlighted in red. It can be observed that after converting the 'for' loop and 'while' loop implementation methods into AST representations, there are significant differences in the number of nodes and branches. When performing word embedding, these differences will still result in different vector representations due to the varying number of nodes and branches. This undoubtedly introduces noise due to different implementation methods, reducing the robustness of the neural network model.

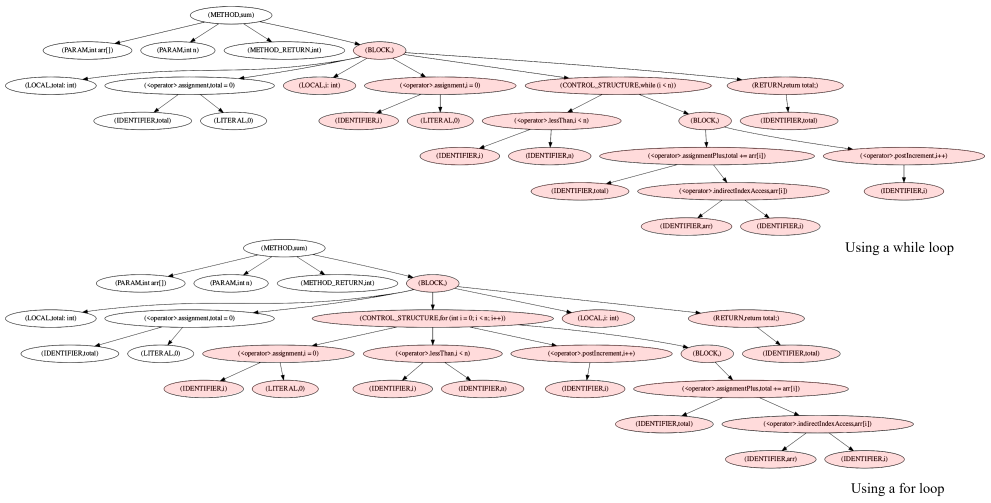
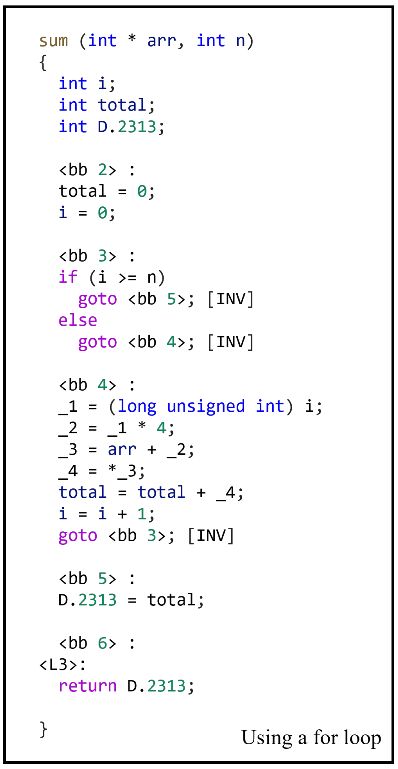


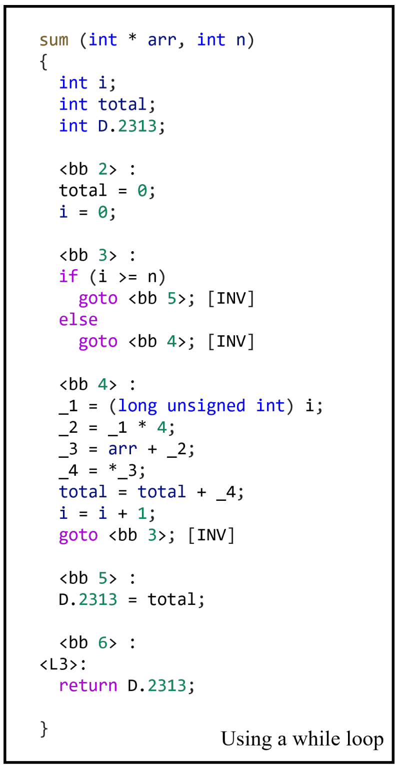
Figure 2. Representing two different implementation methods using AST.

Figure 3 displays the corresponding TAC generated from the source code of the two implementation methods after compilation with GCC. It can be observed that regardless of whether a 'for' loop or a 'while' loop is used, the GIMPLE IR generated by GCC is identical. This suggests that converting the source code into GIMPLE IR representation eliminates the noise impact caused by the differences between the 'for' and 'while' implementation methods.



```
sum (int * arr, int n)
{
  int i;
  int total;
  int D.2313;

  <bb 2> :
  total = 0;
  i = 0;

  <bb 3> :
  if (i >= n)
    goto <bb 5>; [INV]
  else
    goto <bb 4>; [INV]

  <bb 4> :
  _1 = (long unsigned int) i;
  _2 = _1 * 4;
  _3 = arr + _2;
  _4 = *_3;
  total = total + _4;
  i = i + 1;
  goto <bb 3>; [INV]

  <bb 5> :
  D.2313 = total;

  <bb 6> :
<L3>:
  return D.2313;

}
```

Using a for loop

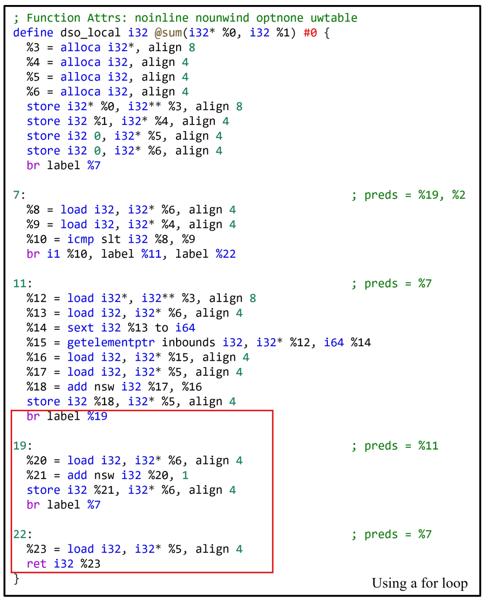


```
sum (int * arr, int n)
{
  int i;
  int total;
  int D.2313;

  <bb 2> :
  total = 0;
  i = 0;

  <bb 3> :
  if (i >= n)
    goto <bb 5>; [INV]
  else
    goto <bb 4>; [INV]

  <bb 4> :
  _1 = (long unsigned int) i;
  _2 = _1 * 4;
  _3 = arr + _2;
  _4 = *_3;
  total = total + _4;
  i = i + 1;
  goto <bb 3>; [INV]

  <bb 5> :
  D.2313 = total;

  <bb 6> :
<L3>:
  return D.2313;

}
```

Using a while loop

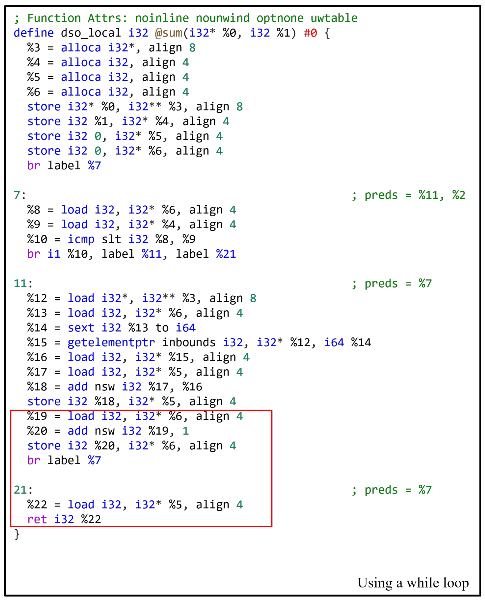
Figure 3. Representing two different implementation methods using GIMPLE IR.

Figure 4 displays the corresponding TAC generated from the source code of two implementation methods after compilation with LLVM (Low Level Virtual Machine). It can be observed that there are still some subtle differences, highlighted in the red boxes in the figure, between the two different implementation methods. These minor differences lead to slight variations in the vectors generated after word embedding. However, compared to directly word-embedding the source code or converting the source code to an AST and then performing word embedding, there is a significant improvement as it eliminates some of the noise impact caused by the differences between the 'for' and 'while' loop implementation methods.



```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @sum(i32* %0, i32 %1) #0 {
  %3 = alloca i32*, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32* %0, i32** %3, align 8
  store i32 %1, i32* %4, align 4
  store i32 0, i32* %5, align 4
  store i32 0, i32* %6, align 4
  br label %7

7:                                                ; preds = %19, %2
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %4, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %22

11:                                               ; preds = %7
  %12 = load i32*, i32** %3, align 8
  %13 = load i32, i32* %6, align 4
  %14 = sext i32 %13 to i64
  %15 = getelementptr inbounds i32, i32* %12, i64 %14
  %16 = load i32, i32* %15, align 4
  %17 = load i32, i32* %5, align 4
  %18 = add nsw i32 %17, %16
  store i32 %18, i32* %5, align 4
  br label %19

19:                                               ; preds = %11
  %20 = load i32, i32* %6, align 4
  %21 = add nsw i32 %20, 1
  store i32 %21, i32* %6, align 4
  br label %7

22:                                               ; preds = %7
  %23 = load i32, i32* %5, align 4
  ret i32 %23
}
```

Using a for loop



```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @sum(i32* %0, i32 %1) #0 {
  %3 = alloca i32*, align 8
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4
  store i32* %0, i32** %3, align 8
  store i32 %1, i32* %4, align 4
  store i32 0, i32* %5, align 4
  store i32 0, i32* %6, align 4
  br label %7

7:                                                ; preds = %11, %2
  %8 = load i32, i32* %6, align 4
  %9 = load i32, i32* %4, align 4
  %10 = icmp slt i32 %8, %9
  br i1 %10, label %11, label %21

11:                                               ; preds = %7
  %12 = load i32*, i32** %3, align 8
  %13 = load i32, i32* %6, align 4
  %14 = sext i32 %13 to i64
  %15 = getelementptr inbounds i32, i32* %12, i64 %14
  %16 = load i32, i32* %15, align 4
  %17 = load i32, i32* %5, align 4
  %18 = add nsw i32 %17, %16
  store i32 %18, i32* %5, align 4
  %19 = load i32, i32* %6, align 4
  %20 = add nsw i32 %19, 1
  store i32 %20, i32* %6, align 4
  br label %7

21:                                               ; preds = %7
  %22 = load i32, i32* %5, align 4
  ret i32 %22
}
```

Using a while loop

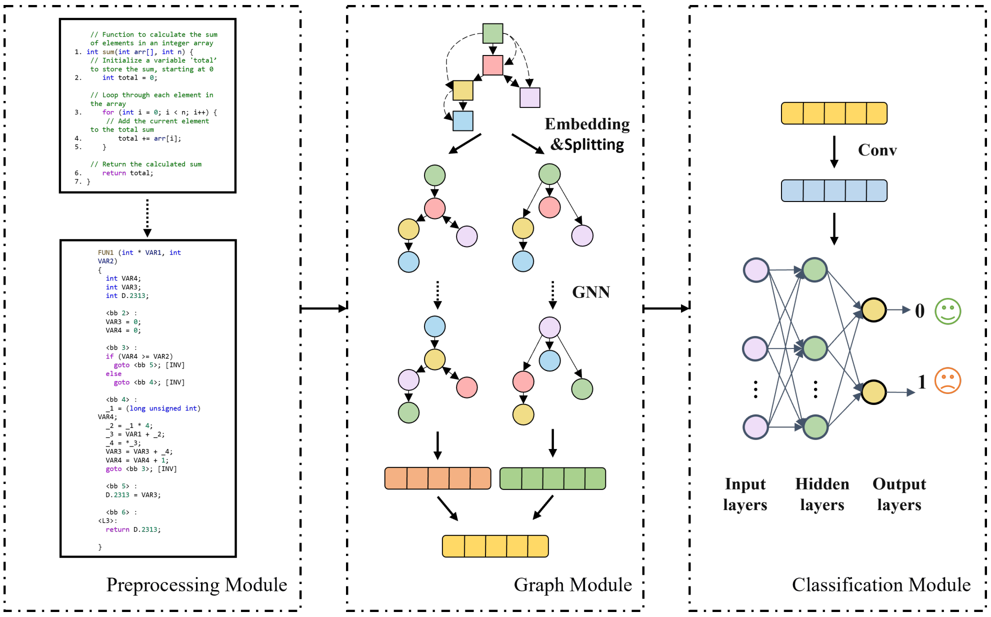
Figure 4. Representing two different implementation methods using LLVM IR.

Upon comparison, it is found that there are significant differences between the 'for' and 'while' implementation methods in both the source code and the AST generated from the source code. In contrast, a TAC can mitigate or even eliminate the noise impact caused by the differences between the 'for' and 'while' implementation methods. Given that GIMPLE IR shows more prominent advantages, this paper opts to convert the source code into GIMPLE IR before feeding it into the neural network.

4. The TACSan Model

In this section, the TACSan model is described. As shown in Figure 5, TACSan consists of three components: the preprocessing module, the graph module, and the classification module. The preprocessing module translates each source code sample into a TAC. The graph module generates the graph structure from the textual TAC and then inputs it into the enhanced graph attention network for training, resulting in a one-dimensional vector. The classification module processes the one-dimensional vector through a convolutional layer for feature aggregation, followed by a fully connected layer for binary classification.

In order to better distinguish between the differences before and after graph embedding, square nodes are used to represent nodes that have not yet undergone graph embedding, with node features being the TAC generated after preprocessing. Circular nodes are used to represent nodes after graph embedding, where the node features are one-dimensional vectors. To indicate that node features change after graph embedding or processing by a GNN, different colors are used within a single graph sample to represent different node features. Color changes in the same nodes across different graph samples signify changes in node features. It is important to note that the same colors between the five graph structures in the example have no connection with each other, merely indicating that the features of the same nodes have changed after passing through the GNN.



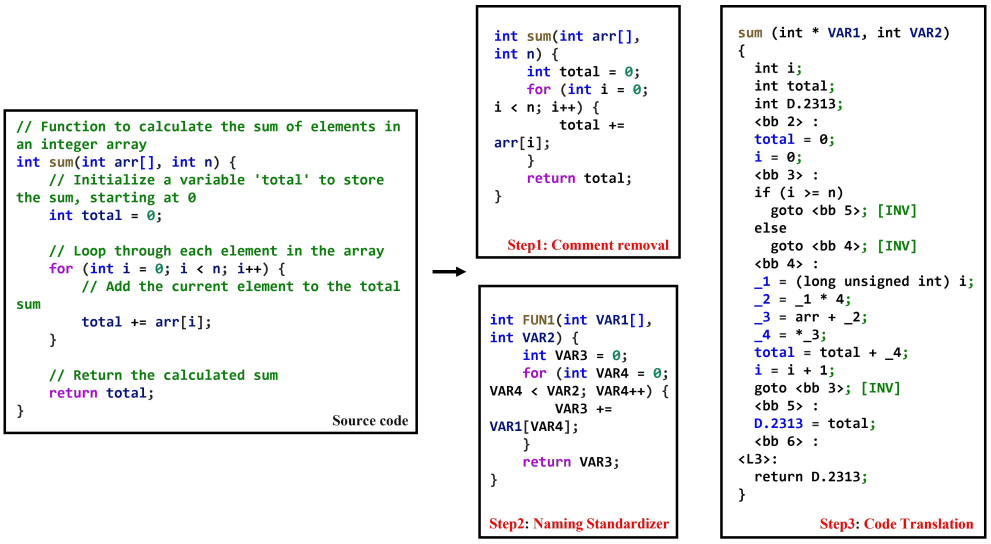
```
// Function to calculate the sum
of elements in an integer array
int sum(int arr[], int n) {
// Initialize a variable 'total'
to store the sum, starting at 0
    int total = 0;

// Loop through each element in
the array
    for (int i = 0; i < n; i++) {
     // Add the current element
to the total sum
        total += arr[i];
    }

// Return the calculated sum
    return total;
}
```

```
FUN1 (int * VAR1, int
VAR2)
{
  int VAR4;
  int VAR3;
  int D.2313;

  <bb 2> :
  VAR3 = 0;
  VAR4 = 0;

  <bb 3> :
  if (VAR4 >= VAR2)
    goto <bb 5>; [INV]
  else
    goto <bb 4>; [INV]

  <bb 4> :
  _1 = (long unsigned int)
VAR4;
  _2 = _1 * 4;
  _3 = VAR1 + _2;
  _4 = *_3;
  VAR3 = VAR3 + _4;
  VAR4 = VAR4 + 1;
  goto <bb 3>; [INV]

  <bb 5> :
  D.2313 = VAR3;

  <bb 6> :
<L3>:
  return D.2313;

}
```

Figure 5. TACSan overall framework.

4.1. Preprocessing Module

The source code acts as the primary bridge for interaction between developers and machines and, thus, is inevitably subject to the individual implementation approaches of the code writers. The preprocessing module is designed to optimize the code and reduce the impact of these varied coding styles to the greatest extent possible. As shown in Figure 6, the figure illustrates how the source code being processed in the preprocessor module changes at each step.



```
// Function to calculate the sum of elements in
an integer array
int sum(int arr[], int n) {
    // Initialize a variable 'total' to store
the sum, starting at 0
    int total = 0;

    // Loop through each element in the array
    for (int i = 0; i < n; i++) {
        // Add the current element to the total
sum
        total += arr[i];
    }

    // Return the calculated sum
    return total;
}
```

```
int sum(int arr[],
int n) {
    int total = 0;
    for (int i = 0;
i < n; i++) {
        total +=
arr[i];
    }
    return total;
}
```

```
int FUN1(int VAR1[],
int VAR2) {
    int VAR3 = 0;
    for (int VAR4 = 0;
VAR4 < VAR2; VAR4++) {
        VAR3 +=
VAR1[VAR4];
    }
    return VAR3;
}
```

```
sum (int * VAR1, int VAR2)
{
  int i;
  int total;
  int D.2313;
  <bb 2> :
  total = 0;
  i = 0;
  <bb 3> :
  if (i >= n)
    goto <bb 5>; [INV]
  else
    goto <bb 4>; [INV]
  <bb 4> :
  _1 = (long unsigned int) i;
  _2 = _1 * 4;
  _3 = arr + _2;
  _4 = *_3;
  total = total + _4;
  i = i + 1;
  goto <bb 3>; [INV]
  <bb 5> :
  D.2313 = total;
  <bb 6> :
<L3>:
  return D.2313;
}
```

Figure 6. Preprocessing module flowchart.

4.1.1. Comment Removal

When handling engineering-level source code, a significant number of comment lines often serve as commentary for the code, aiding developers in understanding complex logic and design intentions. However, when using source code as input for training machine learning models, this advantage can become a challenge. Machine learning models, which learn patterns from data, might mistakenly incorporate comments, especially those with natural language content, into their understanding of the code. This not only interferes with the model's learning of the actual code structure and functionality but also reduces predictive accuracy. Therefore, to ensure the accuracy and reliability of machine learning model training, comment removal is an essential step in the data preprocessing phase. TACSan employs a meticulously designed algorithm and rules to remove or isolate commented parts in the source code, ensuring that the model concentrates on the actual logic and structure of the code without being distracted by comment content.

4.1.2. Naming Standardizer

During the actual code writing process, different developers may adopt their own naming conventions, resulting in a variety of variable, function, and macro names within the code. These naming differences can significantly impact the subsequent feature learning process when utilizing graph neural networks. Consequently, TACSan must standardize these diverse names during the preprocessing stage, for instance, by replacing variables with generic identifiers like VAR1, VAR2, etc., and functions with FUN1, FUN2, etc. Additionally, attention must be given to macro definitions and type aliases in the code, as these special identifiers may introduce further complexity and interfere with the semantic understanding of the original code. They, too, require identification and replacement during the preprocessing phase.

4.1.3. Code Translation

In the actual programming process, standardizing naming conventions minimizes noise interference for the subsequent training of graph neural networks. However, beyond naming conventions, the same functionality can be expressed in various ways. This flexibility, while enriching the diversity of code, also poses challenges for the training of graph neural networks. Therefore, it is necessary to standardize not only variable and function names but also statements. During the preprocessing stage, TACSan reorganizes source code statements into a TAC, which offers several benefits: Firstly, it eliminates differences caused by various code implementation methods, simplifying the model's learning task; secondly, the relatively fixed number of elements per line makes it more concise and standardized than the source code, aiding in noise reduction; finally, it allows the model to represent control flow and data flow information with fine granularity, focusing on the program's logical control structure and data flow without being obscured by specific syntactic details.

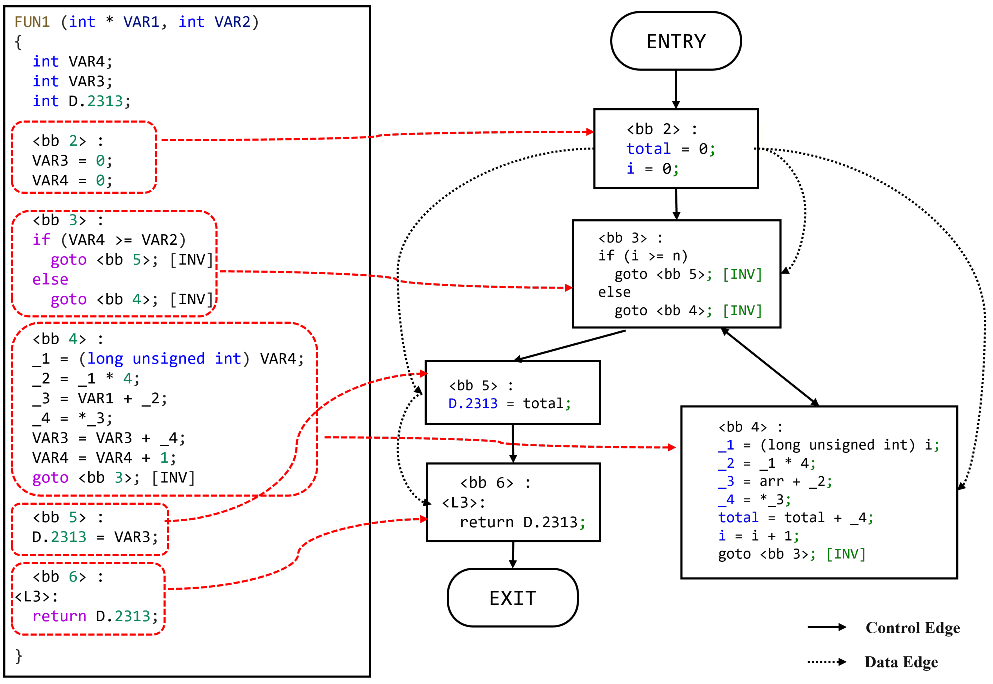
4.2. Graph Module

After the preprocessing module, the original source code is transformed into a TAC representation that exhibits strong regularity, and the amount of redundant information is significantly reduced. However, since the TAC is still in text form, it cannot be directly input into the neural network for training. To address this, the graph module processes the data to ensure it can be accurately handled by the graph neural network.

the code is converted into a IR.
now this IR can be an open source
well available IR or it can be anything
I choose. If i decide to take the TAC route
im gonna get fucked. so i need to properly
see if using TAC has a significant advantage
over LLVM IR

4.2.1. Graph Generation

Graph generation marks the first phase of the graph module, tasked with converting linear structures into graph structures, a critical step that significantly influences the quality of ensuing training samples. Despite being linear, a TAC encompasses implicit data flow and control flow information. By introducing control dependency edges and data dependency edges, the TAC is converted into a graph that encapsulates both data and control flow information. TACSan utilizes basic blocks as graph nodes, employs the TACs within these blocks as node attributes, and leverages control and data flow information for directed edge attributes, constructing a complete heterogeneous graph. Figure 7 provides a straightforward example, illustrating the transformation from the text-bearing TACs to a graph structure equipped with data and control flow information, with solid lines indicating control dependencies and dashed lines indicating data dependencies.



```
FUN1 (int * VAR1, int VAR2)
{
  int VAR4;
  int VAR3;
  int D.2313;

  <bb 2> :
  VAR3 = 0;
  VAR4 = 0;

  <bb 3> :
  if (VAR4 >= VAR2)
    goto <bb 5>; [INV]
  else
    goto <bb 4>; [INV]

  <bb 4> :
  _1 = (long unsigned int) VAR4;
  _2 = _1 * 4;
  _3 = VAR1 + _2;
  _4 = *_3;
  VAR3 = VAR3 + _4;
  VAR4 = VAR4 + 1;
  goto <bb 3>; [INV]

  <bb 5> :
  D.2313 = VAR3;

  <bb 6> :
<L3>:
  return D.2313;

}
```

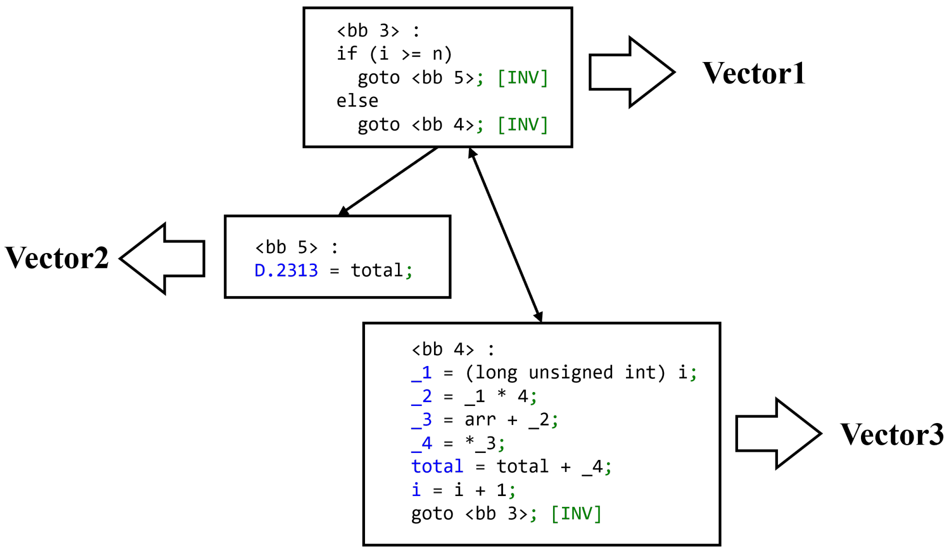
```
ENTRY

<bb 2> :
total = 0;
i = 0;

<bb 3> :
if (i >= n)
  goto <bb 5>; [INV]
else
  goto <bb 4>; [INV]

<bb 5> :
D.2313 = total;

<bb 4> :
_1 = (long unsigned int) i;
_2 = _1 * 4;
_3 = arr + _2;
_4 = *_3;
total = total + _4;
i = i + 1;
goto <bb 3>; [INV]

<bb 6> :
<L3>:
  return D.2313;

EXIT
```

Figure 7. An example shows a TAC converted to a graph structure.

4.2.2. Graph Embedding

Graph embedding is the process of mapping nodes from graph data into a low-dimensional vector space. Following the graph generation phase, the attributes of nodes within the heterogeneous graph remain as high-dimensional text information. This information must be converted into low-dimensional, continuous vectors through graph embedding techniques. Various embedding methods exist, such as One-Hot encoding [24], DeepWalk [25], Word2Vec [26], Doc2Vec [27], and FastText [28]. FastText is particularly adept at capturing the internal morphological features of tokens by utilizing character-level n-grams. For instance, it can recognize the commonality in the characters 'VAR' between 'VAR1' and 'VAR2', a nuance that Word2Vec might overlook. This capability makes FastText the preferred choice for node embedding in this context.

As depicted in Figure 8, some nodes from the aforementioned example are visualized for illustrative purposes. Notably, this paper has elected to retain the <bb> (basic block) tag at the commencement of each node during word vector training. This retention is advantageous as it maintains the programming logic inherent in the sequence of nodes. Post graph embedding, each node's attributes are transformed into a one-dimensional vector suitable for input into the graph neural network for training.



```
<bb 3> :
if (i >= n)
  goto <bb 5>; [INV]
else
  goto <bb 4>; [INV]
```

```
<bb 5> :
D.2313 = total;
```

```
<bb 4> :
_1 = (long unsigned int) i;
_2 = _1 * 4;
_3 = arr + _2;
_4 = *_3;
total = total + _4;
i = i + 1;
goto <bb 3>; [INV]
```

Figure 8. Embedding graph's symbolic representations into vectors.

4.2.3. Graph Training

After obtaining the graph representations from each training sample, TACSan utilizes a GAT (Graph Attention Network) [29], which incorporates an attention mechanism for the training process. Given that GATs are not originally equipped to handle heterogeneous graphs featuring multiple edge types, modifications are necessary to accommodate the training for the various edge types present.

Figure 9 illustrates the improved graph attention network, which primarily involves three key steps. First, the heterogeneous graph is partitioned by edge types into multiple homogeneous subgraphs, allowing the model to focus on specific edge types and effectively process the graph's structural information. Second, the attention mechanism is applied within each subgraph to weigh and aggregate neighboring node features, enabling the model to emphasize the most relevant nodes and enhance feature expressiveness. Lastly, TACSan aggregates node information from subgraphs into a global feature vector representing the graph's features and then aggregates feature vectors from different subgraphs. This step ensures the model integrates local information for a comprehensive understanding.

Through this process, the improved graph attention network more accurately captures the complex relationships between nodes, enabling the model to perform specific tasks more accurately. Consequently, when faced with unseen data or tasks, the model exhibits enhanced generalization capabilities.

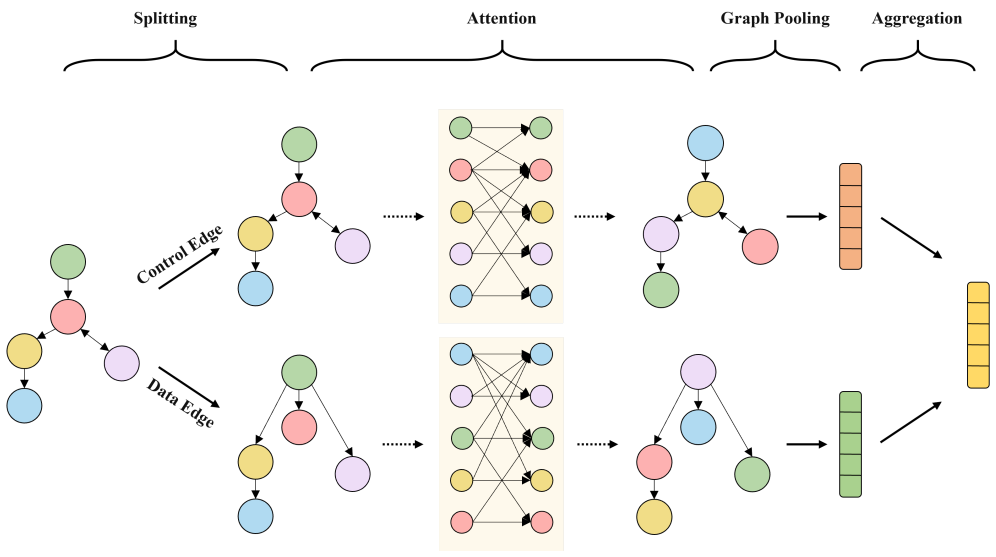


Figure 9. The structure of the graph neural network in TACSan's graph training model.

4.3. Classification Module

After the initial feature extraction in the graph module, TACSan uses the convolution layer to further extract local features from the feature vector. This step enhances the spatial aggregation of features and provides richer contextual information for classification tasks. Subsequently, TACSan introduces an MLP layer for the nonlinear transformation of vectors and performs binary classification in the output layer. The introduction of the nonlinear MLP layer enables the model to learn more complex feature representations, thereby improving classification accuracy.

5. Evaluation

To comprehensively quantify the performance of the model, this section will conduct a comparative evaluation between TACSan and other static detection tools and address the following questions:

RQ1: How does TACSan perform in detecting vulnerabilities in source code?

RQ2: How well does the source code generalize after being transformed into TAC?

RQ3: What is the impact of each module on TACSan?

5.1. Environment

The environment used in this experiment is shown in Table 1.

Table 1. Experimental environment.

Category	Configuration
System OS	Ubuntu 20.04.4 LTS
CPU	Intel® Xeon(R) Silver 4214 CPU @ 2.20 GHz × 48
Graphics	NVIDIA Corporation GP104GL [Quadro P5000]
Memory	128 GiB

In this paper, the version of GCC used in the preprocessing part is 9.4.0, the version of FastText used in the graph network and result classification modules is 0.9.3, the version of PyTorch is 1.13.1, and the version of PyTorch Geometric is 1.6.0.

Referring to previous research [11,14,30–32], in the TACSan graph embedding step, the parameter settings for FastText use the LineSentence object from the gensim library to load the training text, set the dimension of the word vectors to 100, the context window to 3, the minimum n-gram length to 1, the maximum n-gram length to 6, the number of threads for parallel processing to 36, and no word frequency filtering is performed.

After experiments on multiple datasets, as shown in Figures 10 and 11, the final loss function selected is the BCELoss (Binary Cross-Entropy), the optimizer is Adam, the learning rate is set to 0.001, the number of epochs is set to 100, and the batch size is set to 256.

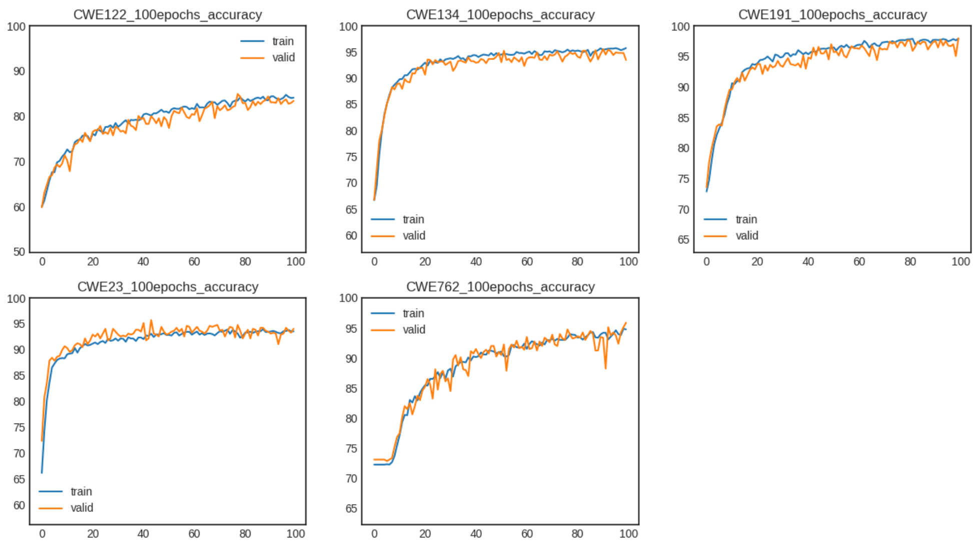


Figure 10. The accuracy of TACSan on the training and test sets of five datasets varies with the change of epochs.

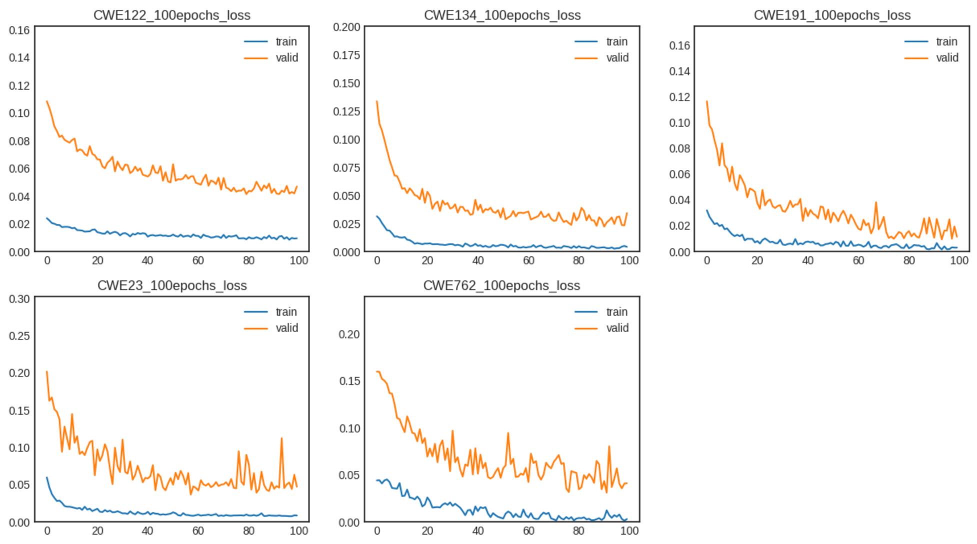


Figure 11. The loss of TACSan on the training and test sets of five datasets varies with the change of epochs.

The specific details of the TACSan model's input and output are shown in Table 2. N1, N2 are the number of nodes in two subgraphs, and B is the batch size. First, GATConv 1 is the first layer in the graph attention network, which receives the input features and processes them through the attention mechanism, making the relationships between nodes more explicit. Then, GATConv 2, as the second layer, continues to process the node features to further extract more complex relational features. After that, the Global Mean Pool performs global average pooling on the node features of the entire graph, condensing the complex information of the graph structure into a fixed-length vector. Next, the processing results of the two subgraphs are stacked and averaged through Stacking and Mean to integrate the output features of these two models. Subsequently, Add new axis adds a new dimension to the feature vector to meet the input requirements of the one-dimensional convolutional layer. The 1D Convolution layer applies convolutional operations to further process the features and extract deeper-level features. Then, the ReLU activation function is applied to the output of the convolutional layer, introducing nonlinear transformations to increase the model's expressive power. Following that, the Flatten layer flattens the processed multi-dimensional features into a one-dimensional vector in preparation for the input of the fully connected layer. Finally, the Fully Connected layer maps the flattened features to the final output categories through linear transformation and converts the output into probability values through the Sigmoid function for the final classification task.

Table 2. Architecture details of the TACSan model.

Layer Name	Parameters	Input Shape	Output Shape
GATConv 1	heads = 4, dropout = 0.4	(N1, 100)	(N1, 64)
ReLU	—	(N1, 64)	(N1, 64)
GATConv 2	heads = 1, dropout = 0.4	(N1, 256)	(N1, 32)
ReLU	—	(N1, 32)	(N1, 32)
Global Mean Pool	—	(N1, 32)	(B, 32)
GATConv 1	heads = 4, dropout = 0.4	(N2, 100)	(N2, 64)
ReLU	—	(N2, 64)	(N2, 64)
GATConv 2	heads = 1, dropout = 0.4	(N2, 256)	(N2, 32)
Global Mean Pool	—	(N2, 32)	(B, 32)
Stacking and Mean	—	(B, 32) + (B, 32)	(B, 32)
Add new axis	—	(B, 32)	(B, 1, 32)
1D Convolution	kernel_size = 3	(B, 1, 32)	(B, 16, 30)
ReLU	—	(B, 16, 30)	(B, 16, 30)
Flatten	—	(B, 16, 30)	(B, 480)
Fully Connected	—	(B, 480)	(B, 2)
Sigmoid	—	(B, 2)	(B, 2)

5.2. Dataset

So far, there has not been a well-designed, widely recognized vulnerability dataset for vulnerability detection. To address this, similar to other related studies [14,15,17,32–35], this experiment draws samples from the SARD (Software Assurance Reference Dataset) of the NIST (National Institute of Standards and Technology) to construct its own vulnerability dataset. The SARD dataset is a publicly accessible collection specifically designed to evaluate the capabilities of static analysis tools. It encompasses a variety of software defects across programming languages such as C, C++, and Java, including repaired code samples where applicable, making it suitable for software security research and tool development. In the data collection phase, this paper identified five types of CWE (CommonWeakness Enumeration) to construct their respective datasets. It should be noted that, for the convenience of feeding into the GNN within TACSan, in the actual construction process, this paper first preprocesses all the source code into a TAC using Python scripts, creating an IR dataset, which is then fed into a GNN for learning and training. After curation, samples dependent on the 'windows.h' header file and those that failed processing were excluded, with the final dataset presented in Table 3. In line with other studies, this paper apportions the dataset into three segments—a training set, a validation set, and a test set—in an 8:1:1 ratio.

Table 3. All datasets and the number of samples with and without vulnerabilities.

Vul Category	All Function	Non-Vulnerable	Vulnerable
CWE122	15,271	9095	6176
CWE134	9604	6402	3202
CWE191	9212	6771	2441
CWE23	5348	2572	2776
CWE762	7978	5768	2210
Total	47,413	30,608	16,805

Below is a brief introduction to each type of CWE.

- **CWE-122 (Heap-Based Buffer Overflow)**: CWE-122 is a common software security vulnerability that occurs when a buffer overflows the allocated memory in the heap section. This overflow can lead to unauthorized code execution, data corruption, or program crashes, among other security issues.
- **CWE-134 (Use of Externally Controlled Format String)**: CWE-134 occurs when a program uses external input as an argument for a format string without proper restrictions or checks. Attackers can exploit CWE-134 to cause buffer overflows, information leaks, or denial-of-service attacks.
- **CWE-191 (Integer Underflow)**: CWE-191 is a programming error that occurs when an integer variable's value exceeds the minimum value it can represent, causing the variable to wrap around to the maximum possible value of that type. This error can lead to logical errors, program crashes, or security vulnerabilities.
- **CWE-23 (Relative Path Traversal)**: CWE-23 is a security vulnerability that occurs when an application uses user-controlled input to construct a file path without properly handling relative paths. This can allow attackers to access or modify unauthorized files, leading to data leaks, data corruption, or system compromise.
- **CWE-762 (Mismatched Memory Management Routines)**: CWE-762 refers to a program calling an incompatible release function when attempting to free memory resources. For example, in C or C++, memory allocated with 'malloc()' should be freed with 'free()', and objects allocated with 'new' should be destroyed with 'delete'. Confusing these different memory management methods can lead to memory corruption, program crashes, or security vulnerabilities, potentially allowing attackers to execute unauthorized code or cause denial-of-service attacks.

5.3. Evaluation Metrics

In order to verify the performance of the model, this paper selects commonly used neural network evaluation criteria to comprehensively assess the model's vulnerability detection capability. Where TP (True Positive) indicates the cases where the model correctly identifies positive class samples as positive. TN (True Negative) indicates the cases where the model correctly identifies negative class samples as negative. FP (False Positive) indicates the cases where the model incorrectly identifies negative class samples as positive. FN (False Negative) indicates the cases where the model incorrectly identifies positive class samples as negative.

Accuracy Accuracy is the proportion of samples correctly predicted by the model out of the total number of samples, which is suitable for situations where the categories are balanced.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Recall Recall is the proportion of samples that are actually positive and correctly predicted as positive by the model, suitable for scenarios where minimizing false negatives is important.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

Precision Precision is the proportion of samples predicted as positive by the model that is actually positive, which is suitable for scenarios where minimizing false positives is important.

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

F1 Score The F1 Score is the harmonic mean of precision and recall, especially suitable for imbalanced datasets. It provides a balanced metric between precision and recall.

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

5.4. RQ1: How Does TACSan Perform in Detecting Vulnerabilities in Source Code?

Currently, there are relatively few open-source static vulnerability detection tools that apply to intermediate language representations. To evaluate TACSan's performance in source code vulnerability detection, this paper compares TACSan with two other well-known static analysis tools, VulDeePecker and Devign, and the results are shown in Table 4.

Table 4. A comparative analysis of TACSan with two other well-known static analysis tools on test sets.

Category	Model	Acc (%)	Rec (%)	Pre (%)	F1 (%)
CWE122	VulDeePecker	87.61	88.10	82.46	85.19
	Devign	82.23	79.92	77.00	78.44
	TACSan	83.38	83.79	77.33	80.43
CWE134	VulDeePecker	89.18	92.19	78.88	85.01
	Devign	91.99	90.00	86.49	88.21
	TACSan	93.44	89.06	91.05	90.05
CWE191	VulDeePecker	92.95	80.33	92.02	85.78
	Devign	94.79	89.34	90.83	90.08
	TACSan	97.94	97.54	94.82	96.16
CWE23	VulDeePecker	92.52	97.84	88.89	93.15
	Devign	91.21	92.81	90.53	91.65
	TACSan	94.02	99.28	90.20	94.52
CWE762	VulDeePecker	91.98	92.09	80.82	86.09
	Devign	89.97	80.47	81.99	81.22
	TACSan	95.86	90.23	94.17	92.16

The experimental results indicate that the tool proposed in this paper performs relatively poorly in detecting vulnerabilities of the CWE122 type, whereas VulDeePecker demonstrates good performance on CWE122 type vulnerabilities. This is attributed to VulDeePecker's focus on buffer error and resource management error types, summarizing relevant C/C++ library/API function calls for buffer error types. To some extent, the advantage of Graph Neural Networks for such vulnerabilities is not evident; hence, TACSan and Devign do not leverage their unique strengths. Apart from this, TACSan outperforms VulDeePecker and Devign in detecting other types of vulnerabilities. Comparatively, on five datasets, the average accuracy improved by 2.08% and 2.89%, and the average F1 score increased by 3.62% and 4.74% compared to VulDeePecker and Devign, respectively. Analysis reveals that one reason is VulDeePecker's sequence-based learning approach, which only has data flow information and neglects control flow information. Another reason is that although Devign is trained with more information, including data and control flow information, along with an AST and natural code sequence data, the excess information also implies noise, leading to decreased results. Considering both Acc and F1, the method proposed in this paper has achieved better outcomes.

5.5. RQ2: How Well Does the Source Code Generalize after Being Transformed into TAC?

To assess the generalization capability following the transformation of source code into a TAC, this paper employs Joern for parsing the source code, constructs an AST, and subsequently inputs it into GAT for training. A comparison of the results with TACSan and those utilizing an AST is presented in Table 5.

The experimental results indicate that when there is a high diversity in the implementation methods within the code, the model performance declines due to the noise from the various implementation methods affecting the generated AST. However, TACSan converts to a TAC during preprocessing, which is less affected by the different implementation methods of the code.

Table 5. Comparing the results from TACSan with those from AST on test sets.

Category	Model	Acc (%)	Rec (%)	Pre (%)	F1 (%)
CWE122	AST	70.16	79.78	60.10	68.55
	TACSan	83.38	83.79	77.33	80.43
CWE134	AST	80.02	77.50	67.39	72.09
	TACSan	93.44	89.06	91.05	90.05
CWE191	AST	79.93	56.56	63.59	59.87
	TACSan	97.94	97.54	94.82	96.16
CWE23	AST	72.34	75.54	72.41	73.94
	TACSan	94.02	99.28	90.20	94.52
CWE762	AST	75.19	19.53	62.69	29.79
	TACSan	95.86	90.23	94.17	92.16

5.6. RQ3: What Is the Impact of Each Module on TACSan?

To verify the impact of each module on TACSan, this paper designed the following experiments. After the comment removal in the preprocessing module, the source code is directly transformed into a graph structure with data and control edges added, and the subsequent processing is the same as TACSan, and this model is named c1. In the graph module, the intermediate language representation sequence after preprocessing is embedded and then fed into the LSTM model for training, and this model is named c2. In the result classification module, without passing through the convolutional layer, the input is directly classified into the MLP layer, and this model is named c3. The test sets' results for accuracy and F1 score are depicted in Figures 12 and 13.

The experimental results indicate that after modifications, the detection effects of the three modules have declined to varying degrees. In the preprocessing module, due to the retention of the most original information of the source code, the model was affected by the noise brought by the habits of the code writers during training. In the graph module, the modified c2 model lost the syntactic structure information. In the result classification module, the convolutional layer can highlight the vulnerability features more effectively, thus achieving better results.

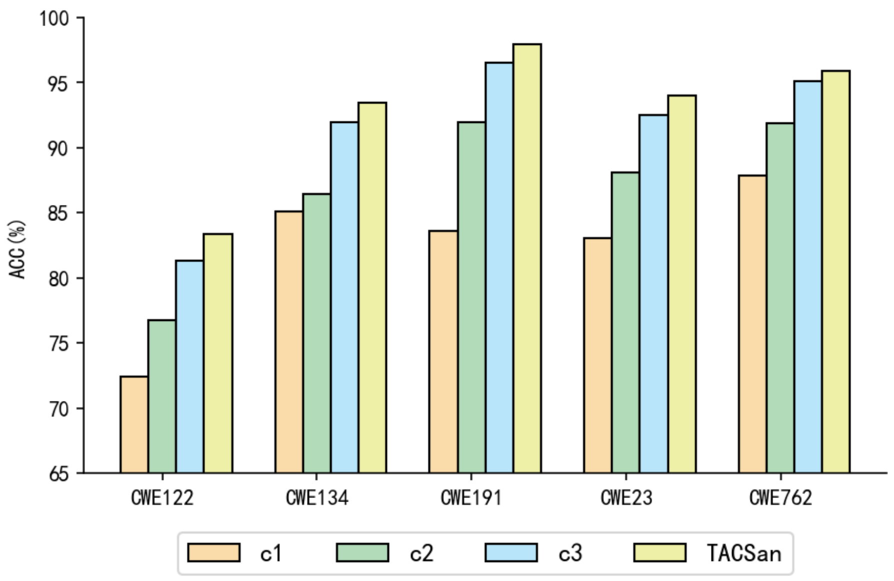


Figure 12. The impact of various modules on the model's accuracy.

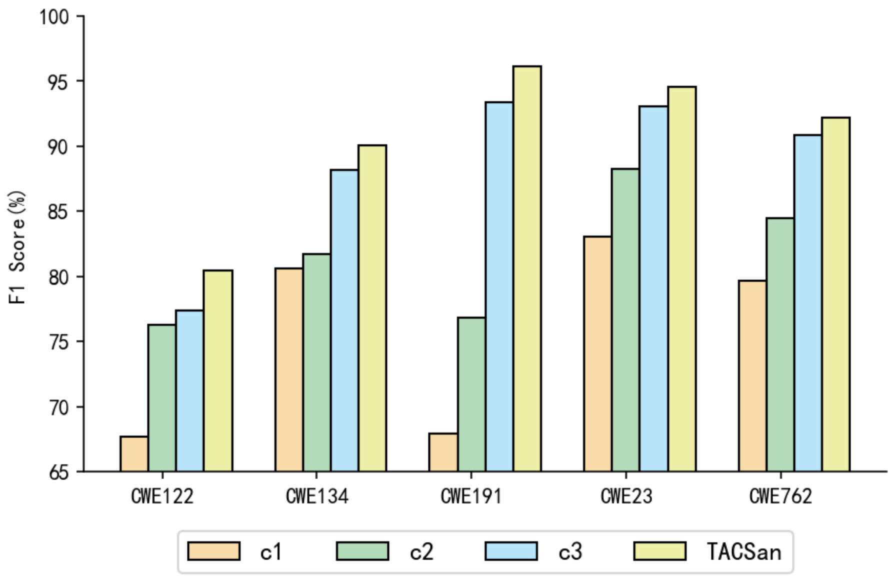


Figure 13. The impact of various modules on the model's F1 score.

6. Limitations

The current work still has certain limitations. This section discusses the six existing shortcomings and potential directions for future improvement.

Firstly, in the preprocessing stage, due to the use of existing compiler tools, it is necessary to ensure the completeness of the function fragments; otherwise, errors will occur during the generation of IR, leading to failed sample processing. Future research may consider tools that can analyze directly at the source code level without the need for compilation.

Additionally, this paper focuses solely on C/C++ programs. Given the compatibility of IR with various high-level languages, which can mask syntactic differences, future studies could include samples from other high-level languages and extend the testing and classification to source code from multiple programming languages.

Thirdly, the conversion to an IR using the compiler alone cannot entirely shield the statement variations caused by different implementation methods. For instance, the IR of ternary expressions versus if statements still shows differences. Future research may consider manually optimizing the IR for specific scenarios to minimize interference from noise due to implementation methods.

Fourthly, this paper has primarily explored the noise generated by different code implementation methods. However, in other aspects, such as the selection of various hyperparameters, this paper has only used the default values of the model without further exploration. In future research, consideration will be given to how to optimize hyperparameters, thereby reducing not only the noise brought about by code implementation methods but also the impact on results due to parameter settings.

Fifthly, the design of the attention mechanism in this paper is quite limited. For the issue that the most primitive attention mechanism cannot be applied to heterogeneous edges, a simple solution is adopted by dividing it into homogeneous graphs. There is still much room for improvement in the attention mechanism for future research work. Currently, many researchers have conducted studies on the attention mechanism [36–39]. How to reasonably improve the attention mechanism and apply it to the model proposed in this paper is a task worth further investigation.

Finally, parallel programming is a fundamental challenge in contemporary computer science. However, in the actual construction of the tools in this paper, there is still much room for improvement in this area. In future research, building upon the work of Yehezkael [40] and others, it may be possible to simplify parallel computing by separating distributed programs into a pure algorithm and a distribution/communication declaration, thereby enhancing the execution efficiency of the tools.

7. Conclusions

In response to the noise introduced into source code due to diverse implementation approaches by code writers, this paper introduces TACSan, a GNN model that leverages IR to mitigate this noise. TACSan preprocesses the function source code fragments and reorganizes the statements into a TAC. It then generates a heterogeneous graph by adding edges that capture data flow and control flow information. Following this, the improved GAT network is trained, and the graph is processed through convolutional and fully connected layers to achieve a binary classification of vulnerabilities and non-vulnerabilities. Tests on the five distinct vulnerability type datasets constructed for this study demonstrate that TACSan significantly surpasses current detection methods in both accuracy and F1 score.

Author Contributions: Conceptualization, D.X.; Data curation, Z.W. and Y.S.; Funding acquisition, D.X.; Methodology, Q.Z.; Project administration, Y.W.; Resources, K.Q.; Software, Q.Z.; Supervision, D.X.; Visualization, Z.W. and K.Q.; Writing—original draft, Q.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Science and Technology on Complex Electronic System Simulation Laboratory, grant number 614201002012204.

Data Availability Statement: The original contributions presented in the study are included in the article. Further inquiries can be directed to the corresponding authors.

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

Abbreviations

The following abbreviations are used in this manuscript:

GNN	Graph Neural Networks
TAC	Three address code
GCC	GNU Compiler Collection
API	Application Programming Interface
LSTM	Long Short-Term Memory
BiLSTM	Bidirectional LSTM
CLSTM	Contextual LSTM
SyVCs	Syntax-based Vulnerability Candidates
SeVCs	Semantics-based Vulnerability Candidates
Bi-GRU	Bidirectional Gated Recurrent Unit
DH-CNN	deep hierarchical convolutional neural network
AST	Abstract Syntax Tree
IR	Intermediate Representation
Tree-LSTM	tree-structured LSTM network
CFG	control flow graph
DFG	data flow graph
CPG	code property graph
PDG	program dependency graph
DDG	data dependency graph
HCG	Hybrid Code Graph
LLVM	Low Level Virtual Machine
GAT	Graph Attention Networks
MLP	Multilayer Perceptron
BCELoss	Binary Cross-Entropy
NIST	National Institute of Standards and Technology
SARD	Software Assurance Reference Dataset
CWE	CommonWeakness Enumeration
TP	True Positive

TN	True Negative
FP	False Positive
FN	False Negative
Acc	Accuracy
Rec	Recall
Pre	Precision

References

1. Habibi, J.; Gupta, A.; Carlsony, S.; Panicker, A.; Bertino, E. MAVR: Code Reuse Stealthy Attacks and Mitigation on Unmanned Aerial Vehicles. In Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems, Columbus, OH, USA, 29 June–2 July 2015; pp. 642–652. [CrossRef]
2. Ahmed, S.J.; Taha, D.B. Machine Learning for Software Vulnerability Detection: A Survey. In Proceedings of the 2022 8th International Conference on Contemporary Information Technology and Mathematics (ICCITM), Mosul, Iraq, 31 August–1 September 2022; pp. 66–72. [CrossRef]
3. Zhang, Y.; Jia, L. A Fuzzy Learning Anti-Jamming Approach With Incomplete Information. *IEEE Commun. Lett.* **2024**, *28*, 1514–1518. [CrossRef]
4. Jia, L.; Qi, N.; Chu, F.; Fang, S.; Wang, X.; Ma, S.; Feng, S. Game-Theoretic Learning Anti-Jamming Approaches in Wireless Networks. *IEEE Commun. Mag.* **2022**, *60*, 60–66. [CrossRef]
5. Zhu, Y.; Lin, G.; Song, L.; Zhang, J. The Application of Neural Network for Software Vulnerability Detection: A Review. *Neural Comput. Appl.* **2023**, *35*, 1279–1301. [CrossRef]
6. Hindle, A.; Barr, E.T.; Gabel, M.; Su, Z.; Devanbu, P. On the Naturalness of Software. *Commun. ACM* **2016**, *59*, 122–131. [CrossRef]
7. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In Proceedings of the 2018 Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018. [CrossRef]
8. Xu, A.; Dai, T.; Chen, H.; Ming, Z.; Li, W. Vulnerability Detection for Source Code Using Contextual LSTM. In Proceedings of the 2018 5th International Conference on Systems and Informatics (ICSAI), Nanjing, China, 10–12 November 2018; pp. 1225–1230. [CrossRef]
9. Saccente, N.; Dehlinger, J.; Deng, L.; Chakraborty, S.; Xiong, Y. Project Achilles: A Prototype Tool for Static Method-Level Vulnerability Detection of Java Source Code Using a Recurrent Neural Network. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), San Diego, CA, USA, 11–15 November 2019; pp. 114–121. [CrossRef]
10. Zou, D.; Wang, S.; Xu, S.; Li, Z.; Jin, H. $\mu\mu$VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Trans. Dependable Secur. Comput.* **2021**, *18*, 2224–2236. [CrossRef]
11. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2022**, *19*, 2244–2258. [CrossRef]
12. Tang, M.; Tang, W.; Gui, Q.; Hu, J.; Zhao, M. A Vulnerability Detection Algorithm Based on Residual Graph Attention Networks for Source Code Imbalance (RGAN). *Expert Syst. Appl.* **2024**, *238*, 122216. [CrossRef]
13. Dam, H.K.; Pham, T.; Ng, S.W.; Tran, T.; Grundy, J.; Ghose, A.; Kim, T.; Kim, C.J. Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. In Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 25–31 May 2019; pp. 46–57. [CrossRef]
14. Feng, H.; Fu, X.; Sun, H.; Wang, H.; Zhang, Y. Efficient Vulnerability Detection Based on Abstract Syntax Tree and Deep Learning. In Proceedings of the IEEE INFOCOM 2020—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Toronto, ON, Canada, 6–9 July 2020; pp. 722–727. [CrossRef]
15. Tian, Z.; Tian, B.; Lv, J.; Chen, Y.; Chen, L. Enhancing Vulnerability Detection via AST Decomposition and Neural Sub-Tree Encoding. *Expert Syst. Appl.* **2024**, *238*, 121865. [CrossRef]
16. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*; Number 915; Curran Associates Inc.: Red Hook, NY, USA, 2019; pp. 10197–10207.
17. Meng, X.; Lu, S.; Wang, X.; Liu, X.; Hu, C. Improving Vulnerability Detection with Hybrid Code Graph Representation. In Proceedings of the 2023 30th Asia-Pacific Software Engineering Conference (APSEC), Seoul, Republic of Korea, 4–7 December 2023; pp. 259–268. [CrossRef]
18. Abdu, A.; Zhai, Z.; Abdo, H.A.; Algabri, R. Software Defect Prediction Based on Deep Representation Learning of Source Code From Contextual Syntax and Semantic Graph. *IEEE Trans. Reliab.* **2024**, 73, 820–834. [CrossRef]
19. Mohammadi, B.; Hong, Y.; Qi, Y.; Wu, Q.; Pan, S.; Shi, J.Q. Augmented Commonsense Knowledge for Remote Object Grounding. *Proc. AAAI Conf. Artif. Intell.* **2024**, *38*, 4269–4277. [CrossRef]
20. Hong, Y.; Rodriguez-Opazo, C.; Qi, Y.; Wu, Q.; Gould, S. Language and Visual Entity Relationship Graph for Agent Navigation. In Proceedings of the 34th International Conference on Neural Information Processing Systems, Red Hook, NY, USA, 6–12 December 2020; NIPS '20; pp. 7685–7696.

21. Khemani, B.; Patil, S.; Kotecha, K.; Tanwar, S. A Review of Graph Neural Networks: Concepts, Architectures, Techniques, Challenges, Datasets, Applications, and Future Directions. *J. Big Data* **2024**, *11*, 18. [CrossRef]
22. Han, J.; Cen, J.; Wu, L.; Li, Z.; Kong, X.; Jiao, R.; Yu, Z.; Xu, T.; Wu, F.; Wang, Z.; et al. A Survey of Geometric Graph Neural Networks: Data Structures, Models and Applications. *arXiv* **2024**, arXiv:2403.00485v1.
23. Khoshraftar, S.; An, A. A Survey on Graph Representation Learning Methods. *ACM Trans. Intell. Syst. Technol.* **2024**, *15*, 1–55. [CrossRef]
24. Wang, Y.; Hou, Y.; Che, W.; Liu, T. From Static to Dynamic Word Representations: A Survey. *Int. J. Mach. Learn.Cyber.* **2020**, *11*, 1611–1630. [CrossRef]
25. Perozzi, B.; Al-Rfou, R.; Skiena, S. DeepWalk: Online Learning of Social Representations. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 24–27 August 2014; KDD '14, pp. 701–710. [CrossRef]
26. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J. Distributed Representations of Words and Phrases and Their Compositionality. In Proceedings of the Neural Information Processing Systems, Lake Tahoe, NV, USA, 5–8 December 2013.
27. Le, Q.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the 31st International Conference on International Conference on Machine Learning—Volume 32, Beijing, China, 21–26 June 2014; ICML '14, pp. II-1188–II-1196.
28. Joulin, A.; Grave, E.; Bojanowski, P.; Mikolov, T. Bag of Tricks for Efficient Text Classification. *arXiv* **2016**, arXiv:1607.01759.
29. Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; Bengio, Y. Graph Attention Networks. *arXiv* **2018**, arXiv:1710.10903.
30. Cao, S.; Sun, X.; Bo, L.; Wu, R.; Li, B.; Tao, C. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks. In Proceedings of the 44th International Conference on Software Engineering, New York, NY, USA, 21–29 May 2022; ICSE '22; pp. 1456–1468. [CrossRef]
31. Li, M.; Li, C.; Li, S.; Wu, Y.; Zhang, B.; Wen, Y. ACGVD: Vulnerability Detection Based on Comprehensive Graph via Graph Neural Network with Attention. In Proceedings of the Information and Communications Security: 23rd International Conference, ICICS 2021, Chongqing, China, 19–21 November 2021; Proceedings, Part I; Springer: Berlin/Heidelberg, Germany, 2021; pp. 243–259. [CrossRef]
32. Zhang, C.; Liu, B.; Fan, Q.; Xin, Y.; Zhu, H. Vulnerability Detection with Graph Attention Network and Metric Learning. *TechRxiv* **2022**. [CrossRef]
33. Chu, Z.; Wan, Y.; Li, Q.; Wu, Y.; Zhang, H.; Sui, Y.; Xu, G.; Jin, H. Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analy, Vienna, Austria, 16–20 September 2024. [CrossRef]
34. Tian, Z.; Tian, B.; Lv, J.; Chen, L. Learning and Fusing Multi-View Code Representations for Function Vulnerability Detection. *Electronics* **2023**, *12*, 2495. [CrossRef]
35. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Softw. Eng.* **2022**, *48*, 3280–3296. [CrossRef]
36. Phan, V.M.H.; Xie, Y.; Zhang, B.; Qi, Y.; Liao, Z.; Perperidis, A.; Phung, S.L.; Verjans, J.W.; To, M.S. Structural Attention: Rethinking Transformer for Unpaired Medical Image Synthesis. *arXiv* **2024**, arXiv:2406.18967.
37. An, D.; Qi, Y.; Li, Y.; Huang, Y.; Wang, L.; Tan, T.; Shao, J. BEVBert: Multimodal Map Pre-training for Language-guided Navigation. *arXiv* **2023**, arXiv:2212.04385.
38. Chen, W.; Hong, D.; Qi, Y.; Han, Z.; Wang, S.; Qing, L.; Huang, Q.; Li, G. Multi-Attention Network for Compressed Video Referring Object Segmentation. In Proceedings of the 30th ACM International Conference on Multimedia, New York, NY, USA, 10–14 October 2022; MM '22, pp. 4416–4425. [CrossRef]
39. Ge, C.; Song, Y.; Ma, C.; Qi, Y.; Luo, P. Rethinking Attentive Object Detection via Neural Attention Learning. *IEEE Trans. Image Process.* **2024**, *33*, 1726–1739. [CrossRef] [PubMed]
40. Yehezkael, R.B.; Wiseman, Y.; Mendelbaum, H.G.; Gordin, I.L. Experiments in Separating Computational Algorithm from Program Distribution and Communication. In *Applied Parallel Comput. New Paradigms for HPC in Industry and Academia, Proceedings of the 5th International Workshop, PARA 2000, Bergen, Norway, 18–20 June 2000*; Sørevik, T., Manne, F., Gebremedhin, A.H., Moe, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2001; pp. 268–278. [CrossRef]