Article
IRC-CLVul: Cross-Programming-Language Vulnerability Detection with Intermediate Representations and Combined Features

Tianwei Lei, Jingfeng Xue, Yong Wang and Zhenyan Liu

School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China; lei_tianwei@bit.edu.cn (T.L.); xuejf@bit.edu.cn (J.X.); wangyong@bit.edu.cn (Y.W.)
* Correspondence: zhenyanliu@bit.edu.cn

Abstract: The most severe problem in cross-programming languages is feature extraction due to different tokens in different programming languages. To solve this problem, we propose a cross-programming-language vulnerability detection method in this paper, IRC-CLVul, based on intermediate representation and combined features. Specifically, we first converted programs in different programming languages into a unified LLVM intermediate representation (LLVM-IR) to provide a classification basis for different programming languages. Afterwards, we extracted the code sequences and control flow graphs of the samples, used the semantic model to extract the program semantic information and graph structure information, and concatenated them into semantic vectors. Finally, we used Random Forest to learn the concatenated semantic vectors and obtained the classification results. We conducted experiments on 85,811 samples from the Juliet test suite in C, C++, and Java. The results show that our method improved the accuracy by 7% compared with the two baseline algorithms, and the F1 score showed a 12% increase.

Keywords: cross-programming language; software vulnerability detection; source code vulnerability detection; combined features; intermediate representation

1. Introduction

Program classification refers to the task of classifying program source code according to the function or semantics of the program [1]. It is an integral part of code intelligence, and it can be regarded as the high-level abstraction of programs based on natural language processing technology and provides new solutions for source-code-based comprehension tasks. Usually, the program classification task is carried out on the source code, which contains rich semantics, and source code is easier to understand and process as a natural language than binary code.

Software vulnerability detection based on source code is one of the crucial tasks of program classification [2]. It aims to extract source code features and vulnerability rules using static analysis and detect whether the sample contains vulnerabilities, which have the advantages of high code coverage and false negative rate. Currently, most source-code-based vulnerability detection research focuses on the vulnerability detection of a single programming language and has achieved high accuracy.

At the same time, research on source code vulnerabilities based on cross programming language is still relatively small and in its infancy. Although different programming languages are used to write programs, code structure and implementation logic are different; when they realize the same function, their purpose for writing is similar, and the security problems they face are similar. Therefore, it is meaningful to detect vulnerabilities across programming languages. It can help further mine the relationship between vulnerability patterns and codes, and promote the development time of patches and the reuse of cross-language codes to help security personnel complete related tasks better. However, the biggest problem in cross-programming-language
vulnerability detection is how to accurately extract the semantic features of programs given the vocabularies of different programming languages. Different programming languages have different tokens, coding logic, and libraries. Even to achieve the same function, different languages may have significant differences in the writing of the source code.

Aiming to solve the above problems, we propose a cross-programming language software vulnerability detection method based on intermediate representation and combined features. We first converted samples from different programming languages to an LLVM intermediate representation (LLVM-IR) to construct the same vocabulary across programming languages. After conversion, we extracted the code sequence and control flow graph, extracting the program’s global code and control structure information with Bi-LSTM and Graph2Vec, generated vectors, and combined them into semantic vectors. Finally, we used the Random Forest model, input the fusion semantic vectors into the model, and performed cross-programming-language vulnerability detection. We conducted experiments on 85,811 samples from the Juliet test suite in C, C++, and Java, and the results show that our method improves the accuracy by 7% compared with the two baseline algorithms, and the F1 score increased by 12%.

In summary, our main contributions are as follows:

1. We propose a cross-programming-language vulnerability detection method based on intermediate representation and combined features, using the LLVM intermediate representation (LLVM-IR) to avoid differences in vocabulary, which constitutes the basis of cross-programming language detection.
2. We propose a combined semantic feature extraction method, using a network based on the attention mechanism to extract code sequence features and capture the global information of code; we used the graph vector generation network to extract the control information of the control flow graph and capture the code structure information.
3. We input the final semantic vector into Random Forest for further information capture and vulnerability pattern training and finally generated a vulnerability classification result. We used three different programming languages and over 80,000 samples to conduct experiments and compared our method with the most advanced programming language semantic capture networks (CodeBERT, InferCode). The result shows that our method is superior in recall, precision, F1 score, and accuracy compared with the baseline algorithms.

The subsequent sections of this paper are structured as follows: Section 2 introduces the related work on software vulnerability detection, code intelligence, and cross-programming language classification; Section 3 introduces the model and the method; Section 4 deals with the experimental design; Section 5 describes the experimental results and analysis; and Section 6 presents the summary of the full text and possible future research directions.

2. Related Work

2.1. Software Vulnerability Detection

Software vulnerability detection is an essential topic in the field of software security and can be divided into static detection and dynamic detection based on whether samples are executed or not [3]. Static detection technology focuses on the information mining and feature construction of programs from the semantics and syntax of code without executing it to identify potential vulnerabilities in the code.

Early research on vulnerability detection and machine learning revolved around building features and classification for software security metrics. Zimmermann et al. [4] conducted a large-scale vulnerability detection empirical study on Windows Vista systems using five classic metrics: code churn rate, complexity, dependency, organization, and coverage. Chowdhury et al. [5] proposed a vulnerability prediction framework with the help of code complexity, coupling, and cohesion indicators. Younis et al. [6] described a function with eight indicators, including the number of code lines, information flow, the number of function calls, and the maximum nesting level of the control structure in the function. They examined 183 vulnerabilities obtained from the national vulnerability database.
With the development of deep learning and natural language processing, vulnerability researchers are now focusing on extracting deeper semantics from source code, combining deep learning semantic models to build semantic vectors, and mining deeper vulnerability patterns of samples. Hazim et al. [7] pre-trained the source code using a natural language processing model and mined the relationship between the vulnerability column and the source code features. Li et al. [8] proposed an automatic vulnerability detection framework based on a hybrid neural network approach in source code, using LLVM-IR as an intermediate representation to reduce the glossary size. They used RNN and CNN for in-depth learning classification and achieved significant results. Tang et al. [9] conducted extensive testing to compare the efficiency of the two most commonly used artificial neural networks, Bi-LSTM and RVFL, for software vulnerability detection. Wu et al. [10] proposed three deep learning models, LSTM, CNN, and CNN-LSTM, for predicting vulnerabilities. They collected function call sequences as features, which represented the execution mode of binary programs, and used deep learning models to predict vulnerabilities.

2.2. Code Intelligence Based on Source Code

Code intelligence, which can be considered a high-level abstraction of natural language processing in programming languages, is mainly focused on source code analysis and is combined with various downstream tasks, such as code bug repair, code generation, and program classification. Program classification is a crucial part of code intelligence, encompassing code cloning detection, code smell classification, program defect, and vulnerability detection.

Wang et al. [11] extracted abstract syntax trees (AST) from the source code, constructed FA-AST by augmenting the original AST with explicit control and data flow edges, and applied two different types of GNNs to measure the similarity of code pairs for code cloning detection. Chen et al. [12] proposed a method to extract and express rich semantics and relationships in error reports in code repair, combining RNN with dependency parsers to automatically extract error entities and their relationships from error reports. Li et al. [13] proposed a search-based automatic program repair technology that combines a neural machine translation-based method with redundant assumptions and sequence-to-sequence learning of correct patches as the source of potential repair statements to automatically repair Java programs. Zhang et al. [14] proposed a new method called DeleSmell to detect code smells based on deep learning models, extracting structural features through LSA and Word2Vec to extract semantic features, constructing CNN branches and GRU to classify the code smell.

2.3. Cross-Program Language Learning

Cross-language learning, which aims to establish learning models between different programming languages, is a nascent area of research. Comparatively speaking, there is relatively little research on cross-programming language learning and it is still in its infancy.

Yahya et al. [15] extracted an abstract syntax tree (AST) by converting the code into an intermediate representation that traverses the sequence to establish the detection of the cloned language. Nafi et al. [16] analyzed source code’s different syntactic features in various programming languages to detect cross-language code clones, using action filters based on cross-language API call similarity to discard non-potential clone situations and achieve good accuracy. Feng et al. [17] proposed a dual-mode pre-training model CodeBERT for programming languages (PL) and natural languages (NL), which supports the universal representation of downstream NL-PL applications, generates universal representations in different programming languages using pre-training models, and supports multiple types of downstream tasks. Bui et al. [18] realized self-supervised learning by predicting the automatic context subtree of AST, generating vectors by training multiple languages and performing well on multiple tasks. Wang et al. [19] proposed a unified abstract syntax tree (UAST) neural network for cross-language program classification, demonstrating promising results on five datasets. Krishnam et al. [20] proposed a neuro-symbolic approach to
identify semantically similar clones in different programming languages using abstract syntax trees (ASTs). Lin et al. [21] introduced a novel method called XCode for cross-language code representation, which involves pre-training multiple source code language models on about 1.5 million code snippets using abstract syntax trees and ELMo-enhanced variational autoencoders. Ullah et al. [22] utilized Program Dependence Graph with Deep Learning (PDGDL) and Term Frequency Inverse Document Frequency (TFIDF) to detect authors of programming source codes written in different languages. Li et al. [23] proposed a lightweight-assisted vulnerability discovery method using a deep neural network (LAVDNN) to identify weaknesses and provide guidance for manual auditing in various programming languages.

3. Method and Model

3.1. Motivation and Framework

The task of detecting software vulnerabilities based on source code is a crucial downstream process in program classification. Its aim is to analyze and categorize samples using natural language processing technology in conjunction with semantic models. Typically, this technology treats the vulnerability’s source code as a document, extracting semantic information from it and converting it into a semantic vector which is then fed into a classification model to obtain the final outcome.

Research into vulnerability detection in a single programming language has yielded high accuracy, but further research is necessary for identifying vulnerabilities across multiple programming languages. Table 1, sourced from Mend.io [24], presents the top three types of vulnerabilities across seven programming languages over the past five years. Although different programming languages have distinct application scenarios that lead to varying types of vulnerabilities in written code, there are still common vulnerabilities shared among them. Cross-language vulnerability detection can identify specified types of vulnerabilities in different languages and establish connections between vulnerability patterns and source codes across programming language boundaries, facilitating the deeper investigation of vulnerabilities and the development of more effective patch mechanisms.

Table 1. Top 3 vulnerabilities with different languages in recent years from Mend.io.

<table>
<thead>
<tr>
<th>Language</th>
<th>The Most Common Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top 1</td>
</tr>
<tr>
<td>C</td>
<td>CWE-119</td>
</tr>
<tr>
<td>C++</td>
<td>CWE-119</td>
</tr>
<tr>
<td>Java</td>
<td>CWE-200</td>
</tr>
<tr>
<td>PHP</td>
<td>CWE-79</td>
</tr>
<tr>
<td>JavaScript</td>
<td>CWE-310</td>
</tr>
<tr>
<td>Python</td>
<td>CWE-200</td>
</tr>
<tr>
<td>Ruby</td>
<td>CWE-79</td>
</tr>
</tbody>
</table>

One major challenge in detecting vulnerabilities across different programming languages is the varying grammar rules and vocabulary encoding features. Successfully bridging this gap and extracting consistent semantic features is crucial to address the issue. To tackle this challenge, we propose a method called IRC-CLVul which utilizes intermediate representations (IR) to convert programs written in different languages into a common vocabulary, reducing the differences in glossary and establishing the foundation for cross-language classification. Our method comprises three stages: IR conversion, feature generation, and classification. In the IR conversion phase, we convert the source code to the LLVM-IR intermediate representation to address the different glossary problem. Next, we construct sequence and graph features in the feature generation phase to extract semantic information, which are then combined to form a final feature vector. Finally, in the classification stage, we use a Random Forest classification model with the feature vector as input and whether the sample has vulnerabilities as the label column for the final classification detection. As in Figure 1.
3.2. Intermediate Representation

Intermediate representation (IR) refers to any form of expression that represents a program from the source language to the target language. It is typically used in the compilation stage, where the compiler converts the source language into IR and then converts the IR into the target language. The IR is a well-structured and clear representation and has a complete set of grammatical structures without changing the semantics of the source code. Therefore, IR can act as a bridge between languages during the compilation process and provide a suitable basis for cross-programming language classification.

As mentioned earlier, an intermediate representation can be any form of expression representing a program during compilation and is not specific to a particular language. Our model uses a specific intermediate representation, LLVM-IR [25], to eliminate the differences between various programming languages. The goal of LLVM-IR was to become a general-purpose IR and is a low-level language used by the LLVM compiler framework. It provides intermediate representations for many high-level languages and serves as a single static assignment (SSA) type of IR to ensure the integrity of the source code semantics.

We first gathered source codes with vulnerabilities written in different programming languages and designated functions with and without vulnerabilities as our positive and negative samples. These codes were then converted into the LLVM-IR intermediate representation using LLVM front-end compilers CLang and JLang. Sample information was derived from LLVM-IR, and semantic embedding vectors were constructed for subsequent classification. Figure 2 displays partial LLVM-IR fragments generated from various codes; it shows two function fragments written in different languages, namely, Example1.java and Example2.c, and their respective LLVM-IR fragments generated using JLang and CLang. Conversion to the LLVM-IR format alleviated vocabulary differences among the original code while retaining the underlying semantics, preparing us well for subsequent classification.
3.3. Semantic Abstract Representation and Vectorization

After translating various programming languages into a common intermediate representation (IR), we proceeded to derive semantic information from the samples using a consistent vocabulary in order to examine the correlation between vulnerabilities and semantic content. To achieve this, we further abstracted the IR and converted the semantic information into vectors, which were then used to generate an embedding of the code for each sample. We utilized the token sequence based on the IR, as well as the more prevalent control flow graph (CFG), to extract program details. Different vectorization methods were applied to generate code-embedding vectors, before splicing them together to create a semantic vector for the final classification model input.

To generate the embedding vector for the IR token sequence, we utilized a Bi-LSTM architecture that leverages the self-attention mechanism for extracting structural information of the intermediate representation. This mechanism, expressed by a formula, addresses the issue of distance-based dependencies in context vocabulary and effectively captures internal links between the representation sentences. Moreover, we employed the self-attention structure to extract internal relationships, which is a key aspect of the Transformer architecture [26] that addresses long-distance dependencies. A Transformer model is composed of \( K \) layers of blocks, which can encode a sequence of instructions into contextual representation at different levels: \( H_k = [h^k_1, h^k_2, \ldots, h^k_n] \), where \( k \) denotes the \( k \)-th layer. For each layer, the layer representation \( H_k \) is computed by the \( k \)-th layer Transformer block \( \text{Transformer}_k(H_{k-1}) \). We calculated the Attention score to reflect the internal relationship of the sentence. The formula of attention mechanism is as follows.

\[
\text{Attention}(Q, K, V) = \text{Softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V
\]

Example 1: Java

```java
public int sum(int a, int b){
    return a + b;
}
```

Example 2: C

```c
int main(int argc, char **argv)
{
    int *a=0;
    a=(int *)malloc(sizeof(int));
    return 0;
}
```

Figure 2. The phase of different source codes inverting to LLVM-IR.
In the formula, $d_k$ represents the hidden representation’s dimensionality, $Q$ denotes the current element, and $K$ denotes all key values in the sequence of the element’s location. Therefore, $Q$ must perform a similarity calculation with every element, ensuring the overall weight coefficient is 1 using the softmax function. Meanwhile, $V$ represents the corresponding weight value of each element, thus calculating the attention weight. $Q$, $K$, and $V$ can be obtained from the previous hidden representation through various linear functions, that is,

$$
\begin{align*}
Q &= H^{l-1}W_Q^l, & Q \in \mathbb{R}^{l \times d} \\
K &= H^{l-1}W_K^l, & K \in \mathbb{R}^{l \times d} \\
V &= H^{l-1}W_V^l, & V \in \mathbb{R}^{l \times d}
\end{align*}
$$

where $H^{l-1}$ represents the $k-1$-th Transformer block. At last, the encoder produces a final contextual representation $H_L = [h_1^L, h_2^L, \ldots, h_n^L]$, which is obtained from the last block.

Once the sentence’s internal dependencies are captured, it is crucial to extract the sentence’s context, also known as context-based semantic information. To achieve this, we utilized Bi-LSTM [27], a deep semantic model commonly employed to capture contextual semantics that belongs to the RNN network. Unlike unidirectional LSTM, Bi-LSTM can efficiently encode information from both forward and backward directions, thereby better capturing the sentence’s semantic dependencies. The LSTM forgetting gate enables cell units to receive varying rates of network information, which can then be forgotten. By reading the values of $x(t)$ and $h(t-1)$ and calculating a value between 0 and 1 using the sigmoid function, the cell unit can determine the percentage of network information it should incorporate into its calculation, and the method is as follows.

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

where $\sigma(\cdot)$ represents the sigmoid function, $W$ is the weight matrix, and $b$ is the bias coefficient. The input gate of the LSTM model decides how much of the input information $x(t)$ should be retained in the cell state $c(t)$ at the present moment. The updated value is determined by the sigmoid function, while a new candidate value vector is created by the tanh function. The formula for both these values is as follows:

$$
\begin{align*}
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{c}_t &= \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t
\end{align*}
$$

Finally, the output gate of the LSTM produces the output value, which depends on the cell state. To determine the cell state output, the sigmoid function is applied. The cell state is first processed using the tanh function and then multiplied by the output of the sigmoid function. The formula for the output value is as follows:

$$
o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t = o_t \odot \tanh(c_t)
$$

where $b$ is the bias in the LSTM cell unit and $W$ is the cyclic weight.

In the latter half of our semantic vector extraction process, we utilized the control flow graph (CFG) to extract the code structure and call information. By employing the Joern [28] auxiliary tool, we directly generated the CFG on LLVM and utilized Graph2Vec [29] to extract the semantic information from the graph, thereby generating the second half of
the semantic vector. CFG forms the basis for our semantic extraction procedure since the internal logic of writing statements and functions remains consistent across various programming languages, barring only the library they call. We extracted the CFG for each code sample as a semantic vector using Graph2Vec, an unsupervised algorithm based on Skip-gram that encodes the entire graph into a vector space. Graph2Vec trains by maximizing the possibility of document prediction words, and the primary algorithmic flow is shown in Algorithm 1.

### Algorithm 1: Graph2Vec

**Input:** $G \leftarrow \{G_1, G_2, ..., G_n\}$: Set of graphs, $G_i \leftarrow (N_i, E_i, \lambda_i)$ for embeddings have to learnt. $D$: Maximum degree of rooted subgraphs to be considered for learning embeddings. $\delta$: the number of embedding dimensions. $e$: the number of epochs. $\alpha$: the learning rate of the model.

**Output:** Matrix of vector representations of graphs $\Phi$.

1. **Initialization:** the matrix $\Phi$
2. for each $e \in e$ do
3.   Sort the $G$ randomly, $X \leftarrow$ SHUFFLE($G$) for each $G_i \in X$ do
4.     for each $n \in N_i$ do
5.       for $d \leftarrow$ to $D$ do
6.         $sgn^{(d)}(\cdot) \leftarrow GetWLSubgraph(n, G_i, d)$
7.         $J(\Phi) \leftarrow -\log(Pr(\Phi^{(d)}| \Phi(G))$ $\Phi \leftarrow \Phi - \alpha \frac{\partial J}{\partial \Phi}$
8.     end
9. end
10. end
11. return $\Phi$

As for the subgraph extraction algorithm function $GetWLSubgraph(n, G_i, d)$ used in it, its specific algorithm flow is as Algorithm 2.

### Algorithm 2: GetWLSubgraph

**Input:** $n$: The root of the subgraph. $G \leftarrow (N, E, \lambda)$: The graph from subgraph. $d$: The degree of neighbours to be considered for extracting subgraph.

**Output:** $sgn^{(d)}$: Rooted subgraph of degree $d$ around the node $n$

1. if $d = 0$ then
2.   $sgn^{(d)} = \lambda(n)$
3. else
4.   $N_n \leftarrow \{n' | (n, n') \in E\}$
5.   $M_n^{(d)} \leftarrow \{GetWLSubgraph(n', G, d - 1)\}$
6.   $sgn^{(d)} \leftarrow sgn^{(d)} \cup GetWLSubgraph(n', G, d - 1) \oplus sort(M_n^{(d)})$
7. end
8. return $sgn^{(d)}$

We set the two parts to generate 64-dimensional semantic vectors and realized the confirmation of the final semantic vector through vector splicing, which can be described as:

$$\text{vector}_{all} = \text{contact}(\text{vector}_{sequence}, \text{vector}_{CFG})$$  \hspace{1cm} (6)

### 3.4. Classification

After generating semantic vectors, the method proceeds to the final classification stage. The main objective of this stage is to use the previously generated vectors, coupled with
machine learning classification algorithms, to detect vulnerabilities. During this stage, the 128-dimensional semantic vector created in the feature construction stage is inputted into the Random Forest model for classification. Random Forest was selected due to being an effective and simple model that is capable of processing high-dimensional vectors in machine learning, and is widely used. The bootstrap method is employed to extract \( k \) samples from the original training sample set \( N \), where a corresponding decision tree model is created for the \( k \) extracted samples. Finally, the results of the \( k \) samples are voted on, and the final classification is determined based on the majority rule. Its classification decision function is as follows:

\[
H(x) = \arg \max_y \sum_{i=1}^{k} I[h_i(x) = Y]
\]  

Among them, \( H(x) \) represents the aggregate classification model, \( h_i \) refers to the individual decision-making classification models, and \( Y \) denotes the sample label column indicating the presence of vulnerabilities. We extract self-contained sample labels from the dataset, labeling samples with vulnerabilities as 1 and those without as 0. These fused semantic vectors and labels, comprising combined features, are fed into a Random Forest model. The output of this model is then utilized to determine the presence of vulnerabilities in the samples. Table 2 displays the optimal parameters achieved by our model during training.

Table 2. The parameters of the classifier.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_estimators</td>
<td>The number of decision trees.</td>
<td>100</td>
</tr>
<tr>
<td>Criterion</td>
<td>The division standard of the node.</td>
<td>Gini</td>
</tr>
<tr>
<td>Max depth</td>
<td>The maximum depth of the decision tree.</td>
<td>10</td>
</tr>
<tr>
<td>Min samples leaf and Min</td>
<td>The minimum samples contained in the leaf node.</td>
<td>2</td>
</tr>
<tr>
<td>samples split</td>
<td>The maximum number of features considered when</td>
<td></td>
</tr>
<tr>
<td></td>
<td>building the decision tree</td>
<td></td>
</tr>
<tr>
<td>Max features</td>
<td></td>
<td>auto</td>
</tr>
</tbody>
</table>

4. Experiments Design

This section will introduce the problems we explored in the experiment, the dataset, the experimental baseline, and the evaluation indicators.

4.1. Research Questions

Our motivation for designing experiments was to explore the effects of our method in various aspects. Therefore, we asked the following questions and conducted experimental investigations around them.

RQ1: How does our proposed method compare to the baseline algorithms in cross-programming languages' vulnerability detection?

RQ1 is the most crucial question; it explores how our method compares with the baseline methods. For the baseline algorithms, we explored cross-programming language representation and classification methods, providing corresponding names and descriptions in subsequent chapters. We discuss RQ1 in detail in Section 5.1.

RQ2: Is our proposed intermediate representation effective?

RQ2 aims to explore whether our method truly alleviates the challenge of cross-programming languages—significant differences in probability distributions between different programming languages. Our method converts various programming languages into intermediate representations to build the same vocabulary. We conduct experiments
around RQ2 to compare the impact of using and not using intermediate representations in cross-programming-language vulnerability detection. We also consider whether the original semantic information of programming languages is lost when converting them into intermediate representations. When using IR, the accuracy of single-programming-language vulnerability detection can infer the loss of semantic information upon conversion to IR. We discuss RQ2 in Section 5.2.

**RQ3:** Is our proposed combined feature generated using the fusion method effective?

In RQ3, we explore how using the feature fusion method impacts our proposed method’s accuracy. We compared model accuracy when using a single feature versus combined features. We provide further discussion on RQ3 in Section 5.3.

### 4.2. Datasets and Evaluation Metrics

We collected vulnerability samples in three programming languages: C, C++, and Java, all from the Juliet Test Suite. This suite was released by NIST in 2010 and updated to version 1.3 in 2018 [30]. The number of vulnerabilities and samples varies across the three languages, and we selected these languages based on the number of specific vulnerability types, language, common vulnerability types, and the number of samples. Table 3 displays these results, including the number of common and unique vulnerability types, as well as the number of vulnerabilities common to all three languages. We will use these common vulnerabilities and unique vulnerabilities for each language as our dataset, which we will further specify in the following chapters.

<table>
<thead>
<tr>
<th>Number of Common Vulnerability Types</th>
<th>Number of Vulnerabilities</th>
<th>Number of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C++</td>
<td>Java</td>
</tr>
<tr>
<td>C</td>
<td>-</td>
<td>40</td>
</tr>
<tr>
<td>C++</td>
<td>40</td>
<td>-</td>
</tr>
<tr>
<td>Java</td>
<td>26</td>
<td>23</td>
</tr>
</tbody>
</table>

During our training, we selected samples randomly for each iteration; 80% were assigned as the training set, 10% were allotted to the verification set, and another 10% were dedicated to the test set. These steps were taken to facilitate the exploration of various inquiries. The model’s evaluation metrics included accuracy, precision rate, recall, and F1 score, with the formula as follows:

\[
F1\text{score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

The precision and recall are represented as follows.

\[
\text{Precision} = \frac{TP}{TP + FP}
\]

\[
\text{Recall} = \frac{TP}{TP + FN}
\]

where TP represents the positive samples predicted by the model as positive classes, TN means the negative samples predicted by the model as negative classes, FP represents the negative samples predicted by the model as positive, and FN are the positive samples predicted by the model as negative classes.

### 4.3. Baseline

We evaluated our method against the industry-leading cross-language code learning algorithms CodeBERT and InferCode.
CodeBERT [17] is a language model capable of processing natural languages and programming languages, such as Python, Java, and JavaScript. It can capture semantic connections between natural language and code, producing a general representation suitable for NL-PL understanding tasks like natural language code search, and for generation tasks such as code documentation.

As our focus is generating reasonable cross-language embeddings for vulnerability detection and classification, we trained the models using the same dataset, inputting semantic vectors of the same recommended dimension into a Random Forest model. We optimized the model for best performance, comparing our results with those of CodeBERT and InferCode.

5. Results and Analysis

In this subsection, we will focus on the three exploration questions (RQs) proposed in Section 4.1 and demonstrate the effect of our model from multiple perspectives.

5.1. The Accuracy of the Model

For the questions raised in RQ1, we compare our method with the baseline algorithms mentioned in Section 4.3, use the evaluation indicators mentioned in Section 4.2 as the standard, and use the common vulnerabilities of all languages as samples, and the average result of all vulnerabilities as shown in Table 4. Our method performs best among the four selected evaluation metrics. Specifically, our method achieves a 6% improvement in accuracy and an 8% improvement in F1 score for the CodeBERT method, while for the Infercode method, we achieve a 7% improvement in accuracy and a 12% improvement in F1 score.

Table 4. The results of the methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 score</th>
</tr>
</thead>
<tbody>
<tr>
<td>ours</td>
<td>0.9585</td>
<td>0.8669</td>
<td>0.9526</td>
<td>0.9077</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.8976</td>
<td>0.7642</td>
<td>0.8880</td>
<td>0.8215</td>
</tr>
<tr>
<td>InferCode</td>
<td>0.8864</td>
<td>0.7493</td>
<td>0.8302</td>
<td>0.7877</td>
</tr>
</tbody>
</table>

Figure 3 presents the performance results for cross-language vulnerability detection on common vulnerabilities. Our method outperforms the two baseline algorithms in terms of accuracy and F1 score, and exhibits improved stability across different types of vulnerabilities. The accuracy and F1 score of our model remain relatively stable for common vulnerabilities in C, C++, and Java, as shown in Figure 3a. Figure 3b–d show the results of common vulnerabilities in pairs of languages. In the case of C and C++ vulnerabilities, all three methods have similar performances, but our proposed model achieves the highest accuracy, precision, and F1 score, and the recall score is only 1% lower than CodeBERT. For C++ and Java vulnerabilities, our method achieves an accuracy and F1 score above 0.9, showing better performance than the baseline methods.

In terms of the two baseline algorithm models, the CodeBERT model demonstrates high accuracy and performs best on individual vulnerabilities. However, it also displays inevitable volatility in the F1 score. On the other hand, while the InferCode model yields slightly lower accuracy than the other two models, it exhibits more stable features regarding the F1 score. As a variation of BERT, CodeBERT boasts a vast NL-PL network and has become a benchmark algorithm for program classification. Nevertheless, the embedding utilized by CodeBERT in generating semantic vectors may lead to insufficient capture of the code’s semantic features. InferCode, on the contrary, performs well in classification by learning the subtree features of the AST and undergoing further training across the programming language corpus. Nevertheless, its drawback is that there is a limit to the extracted vector dimension, and it has certain restrictions on the semantic capture of long-sequence programs. Our method, which starts from the sequence and CFG and possesses a semantic capture mechanism from both the global program and local code, enhances accuracy to a certain extent and addresses this issue.
The result of RQ1: The proposed method is higher than the compared baseline algorithm in terms of the proposed evaluation indicators; the accuracy is 6% to 8% higher than the baseline, and the F1 score is 8% to 12% higher.

Figure 3. The model results on cross-language common vulnerabilities. (a) The results on all common vulnerabilities of 3 languages. (b) The results on common vulnerabilities of C and C++. (c) The results on common vulnerabilities of C and Java. (d) The results on common vulnerabilities of C++ and Java.

5.2. The Effect of IR

To investigate RQ2, we conducted experiments to assess the effectiveness of intermediate representations in reducing the differences between programming languages. We compared the performance of two methods: one that extracts programming language sequence features and CFG graphs directly (UnIR method), and another that uses intermediate representations in cross-language vulnerability detection. We also assessed the accuracy of feature extraction based on the intermediate representation in a single programming language and investigate whether converting source code to intermediate representation leads to a loss of semantic information.

Figure 4 presents the results of our method compared to the UnIR method in cross-programming-language vulnerability detection. Our proposed method, Origin, which uses intermediate representations, performs better in terms of accuracy and F1 score. In a cross-language vulnerability detection involving three programming languages as shown in Figure 4a, the intermediate representation method improves accuracy by 12% and F1 score by 7% on average compared to the UnIR method. In pairwise vulnerability detection, the use of an intermediate representation to alleviate vocabulary differences also improves accuracy. Figure 4b–d show the different results examining whether IR was used on common vulnerabilities in pairs of languages. In the cross-programming-language vulnerability detection of two languages, the method of using intermediate representation
to alleviate the vocabulary also improves vulnerability detection accuracy. In C and C++, utilizing intermediate representation enhances precision by an average of 5% and increases the F1 score by 7%. In C–Java and C++–Java, our approach elevates accuracy by 5% and 4%, respectively, and improves the F1 score by an average of 6% and 3%. These findings authenticate the effectiveness and necessity of applying intermediate representations in detecting vulnerabilities across programming languages, while also verifying the impact of diverse vocabularies in reverse engineering.

Figure 4. The results of IR comparison. (a) The results on all common vulnerabilities of 3 languages. (b) The results on common vulnerabilities of C and C++. (c) The results on common vulnerabilities of C and Java. (d) The results on common vulnerabilities of C++ and Java.

Figure 5 illustrates the performance of two methods in single-language vulnerability detection. Figure 5a–d demonstrate the effects of incorporating an intermediate representation (IR) on accuracy, precision, recall, and F1 score separately. For each programming language, we compared the detection rates of the methods with and without IR. The term Without IR refers to a single-language detection method that does not employ an intermediate representation, while With IR denotes a method that does. The two extracted features differ in sequence depending on the presence of an intermediate representation. From the average point of view, in the single-language vulnerability detection of C, C++, and Java, the accuracy of using the intermediate representation is reduced by 1%, 1%, and 1%, respectively, and the F1 score is affected by 1%, 2%, and 2%.

In general, selecting an intermediate representation means that some semantic information will be lost during the conversion process from source code to LLVM-IR, resulting in an average reduction of about 1% in single-language vulnerability detection. However, when detecting cross-programming-language vulnerabilities, using intermediate representation can improve detection rates by an average of 10% compared to directly analyzing the source
code. Therefore, based on our experiments comparing single-programming-language and cross-programming-language vulnerability detection using intermediate representation, it is feasible to use intermediate representation to address cross-programming-language vocabulary issues and improve the effectiveness of vulnerability detection.

**Figure 5.** The results of IR comparison on single-language vulnerabilities. (a) The accuracy. (b) The precision. (c) The recall. (d) The F1 score.

**RQ2 results:** Our proposed method of using intermediate representation as the basis to bridge the gap between programming languages has proved effective in significantly improving the accuracy of cross-language vulnerability detection. Our IR diminishes the impact of different vocabularies in such detection and enhances accuracy without any discernible loss of precision.

5.3. The Effect of Combined Feature

To address the RQ3 problem, we conducted an experiment using only program sequence and control flow graph (CFG) features to detect cross-programming-language vulnerabilities. The experimental dataset used was a set of common vulnerabilities, which allowed us to compare the performance of different feature combinations. Table 5 presents the results of three methods: IRC-CLVul, our original approach that combines sequence and CFG features; S-CLVul, a method that only employs sequence features; and G-CLVul, a method that only extracts CFG features.

Table 5 shows that IRC-CLVul, using combined features, outperforms the other methods on all evaluation indicators except for the C–C++ datasets. The combined feature with fusion method improves performance by 3% to 11% on the three cross-programming-language detection datasets, and also improves accuracy and F1 score on cross-programming vulnerability detection samples of other languages, except for the common vulnerabilities of C and C++ detection. On C–C++, while the precision and F1 score of combined features are slightly lower than those obtained using graph features,
they are higher than those obtained with sequences alone. This demonstrates the practicality of our combined feature. Compared to using sequence features and graph features alone, combined features generated using the fusion method can more comprehensively extract semantic information in samples, mine vulnerability patterns, and provide better vulnerability detection results. 

Table 5. The result of combined features on different cross-language datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Method</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>IRG-CLVul</td>
<td>0.9585</td>
<td>0.8669</td>
<td>0.9526</td>
<td>0.9077</td>
</tr>
<tr>
<td></td>
<td>IR-CLVul</td>
<td>0.8458</td>
<td>0.8261</td>
<td>0.8974</td>
<td>0.8603</td>
</tr>
<tr>
<td></td>
<td>G-CLVul</td>
<td>0.9250</td>
<td>0.7881</td>
<td>0.9134</td>
<td>0.8461</td>
</tr>
<tr>
<td>C–Java</td>
<td>IRG-CLVul</td>
<td>0.9343</td>
<td>0.8789</td>
<td>0.9175</td>
<td>0.8978</td>
</tr>
<tr>
<td></td>
<td>IR-CLVul</td>
<td>0.8994</td>
<td>0.7913</td>
<td>0.8581</td>
<td>0.8233</td>
</tr>
<tr>
<td></td>
<td>G-CLVul</td>
<td>0.9274</td>
<td>0.8517</td>
<td>0.9060</td>
<td>0.8780</td>
</tr>
<tr>
<td>C++–Java</td>
<td>IRG-CLVul</td>
<td>0.9573</td>
<td>0.8727</td>
<td>0.9118</td>
<td>0.8918</td>
</tr>
<tr>
<td></td>
<td>IR-CLVul</td>
<td>0.8737</td>
<td>0.7943</td>
<td>0.8581</td>
<td>0.8250</td>
</tr>
<tr>
<td></td>
<td>G-CLVul</td>
<td>0.9128</td>
<td>0.8231</td>
<td>0.8565</td>
<td>0.8395</td>
</tr>
<tr>
<td>C–C++</td>
<td>IRG-CLVul</td>
<td>0.9247</td>
<td>0.7487</td>
<td>0.8695</td>
<td>0.8046</td>
</tr>
<tr>
<td></td>
<td>IR-CLVul</td>
<td>0.8761</td>
<td>0.7907</td>
<td>0.8190</td>
<td>0.8046</td>
</tr>
<tr>
<td></td>
<td>G-CLVul</td>
<td>0.8610</td>
<td><strong>0.8143</strong></td>
<td>0.8429</td>
<td><strong>0.8283</strong></td>
</tr>
</tbody>
</table>

It is worth noting that the performance of S-CLVul, which extracts feature information from instruction sequences, is consistently lower across all datasets than that of G-CLVul, which uses graph features. This suggests that the feature information derived from instruction sequences is less effective in detecting vulnerabilities, compared to that obtained from graphs. This disparity can be attributed to differences in the amount of semantic and structural information accessible through the two types of features. Instruction sequence features tend to capture the relationships between instructions and the global information of the code, while control flow graph features perform better at capturing local details such as loops and branches. Our approach combines both types of features to comprehensively consider the semantic information of a sample, thereby establishing a better classification relationship with the label column.

**RQ3 results**: The combined feature and fusion method we proposed can better extract the internal information of the program from both local and global perspectives. Compared with the method of using two features alone, it can better improve the model effect.

5.4. Analysis and Comparison

In the aforementioned RQs, we presented the model’s effectiveness and features from multiple perspectives. In this section, we will discuss the impact of cross-programming-language vulnerability classification, and summarize the phenomena and conclusions observed in the experiments. Firstly, we highlight the effectiveness of our proposed model as demonstrated in RQ1, where it achieved the best performance on all four test metrics. In RQ2 and RQ3, we validate that the intermediate representation (IR) can effectively reduce the gap between different languages, while accurately representing the internal information of the program, thereby laying the foundation for cross-programming-language vulnerability detection. Lastly, we confirmed that using combined features is more effective than using a single feature.

Next, we compare our taxonomy with other cross-programming-language classifications. Owing to the limited research results in this area, we mainly compare our findings with other cross-programming-language classifications. As stated earlier, the most severe challenge in cross-programming-language classification is addressing the differences in vocabulary. Using intermediate representations to learn program code is a well-established approach, which we demonstrate to be effective in our experiments. Some studies directly
extract code features and resolve the vocabulary differences through API or vocabulary alignment of the same corpus. Although this method achieves higher accuracy, it requires significant efforts in the early stages and takes a longer processing time. Some studies adopt cross-language processing methods from the natural language processing (NLP) field, utilizing XLR-based models to process programming languages, and have also achieved good results.

Next, we move on to feature extraction. Our approach involves using IR sequence and CFG image information to extract features. The IR sequence primarily extracts local information, while the CFG image information mainly extracts global information. We chose LLVM-IR sequence information because it retains semantic information and automatically replaces function name keywords, providing effective program semantic information while establishing a cross-language vocabulary. Additionally, a Bi-LSTM based on the attention mechanism was utilized to quickly extract global information. To extract local structural information, we employed Graph2Vec, which converts the full graph into a vector to extract the control flow graph. Finally, we spliced the features, merging the learned global and local information to obtain better classification results. AST and other graph-based forms are also common means of semantic information extraction, but we ultimately chose to use code sequences and control flow graphs to ensure optimal program information extraction.

As for the limitations of our method, there are a few areas of concern. Firstly, while our feature extraction method using LLVM-IR and graph features contributes to improved accuracy, it may be time-consuming and less efficient when dealing with large programs. Secondly, we currently only use the Juliet suite, a commonly used test set in the field of vulnerabilities, which has achieved certain results. However, we have yet to test it in programs containing vulnerabilities, and while some research and test set creators have noted that it has good generalization ability, our future research direction will involve experimenting with real-world test sets. Lastly, our current focus is on the detection performance of common types of vulnerabilities across programming languages, and we plan to expand our research to include different types of vulnerabilities in the future.

6. Conclusions and Future Work

Cross-programming-language vulnerability detection is a crucial task in program classification. To enhance our ability to tackle this issue, we propose the IRC-CLVul method, which utilizes intermediate representations and combined features. First, we convert various programming languages into LLVM-IR intermediate representation to overcome the problem of different vocabularies. Then, we extract semantic and structural information from the program’s statement sequence and control flow graph, merging them into the final semantic vector. Finally, we employ a Random Forest model to perform the classification. Our experiments demonstrate that IRC-CLVul outperforms other baseline algorithms and achieves excellent detection outcomes for cross-programming-language vulnerabilities. In the future, we will investigate various types of cross-programming-language vulnerabilities based on source code.

Author Contributions: Conceptualization, T.L. and J.X.; methodology, T.L. and Y.W.; software, T.L. and Z.L.; validation, T.L., J.X. and Y.W.; data curation, T.L. and Z.L.; writing—original draft preparation, T.L. and Y.W.; writing—review and editing, T.L. and Z.L.; visualization, T.L. and Z.L.; supervision, J.X. and Y.W.; project administration, J.X. and Y.W.; funding acquisition, J.X. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (No. 62172042) and the Major Scientific and Technological Innovation Projects of Shandong Province (2020CXGC010116).

Data Availability Statement: Not applicable.

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


5. Chowdhury, I.; Zulkernine, M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. J. Syst. Archit. 2011, 57, 294–313. [CrossRef]


25. LLVM. Available online: https://llvm.org/ (accessed on 1 January 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.