The Journal of Systems and Software 232 (2026) 112658

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

PreMulBVD: A pretraining-based multi-modal binary vulnerability detection framework[☆]

Chenliang Xing[a], Xiaoyuan Xie[a,*], Qi Xin[a,b,c,*], Gong Chen[a]

[a] School of Computer Science, Wuhan University, Wuhan, China
[b] Hubei Luojia Laboratory, Wuhan, China
[c] State Key Lab. for Novel Software Technology, Nanjing University, Nanjing, China

ARTICLE INFO

Keywords:
Vulnerability detection
Binary program
Pre-trained model
Decompile
Deep learning

ABSTRACT

Modern software contains vulnerabilities. These vulnerabilities can be exploited for attacks and thus pose severe threats to the software security. Static binary vulnerability detection is crucial at reducing the threats, as it can find vulnerabilities in program binaries to repair and thus reduce the risk of exploitation for attacks. Current approaches are significantly limited in that they do not leverage the high-level semantics and the code structure effectively for vulnerability detection and that the models they use have weak generalizability.

To address these limitations, we propose *PreMulBVD* for static binary vulnerability detection. *PreMulBVD* is novel in that it extracts and combines the pseudo-code containing the high-level semantics and the Abstract Syntax Trees encoding the code structure for vulnerability detection. Furthermore, by using a pre-trained model, *PreMulBVD* extracts robust patterns from the pseudo-code and ASTs, significantly improving the detection accuracy. The result of our evaluation demonstrates the effectiveness of *PreMulBVD*, as it shows that *PreMulBVD* achieves high accuracy, precision, and recall (all over 0.97) for vulnerability detection and outperforms the state-of-the-art approaches.

1. Introduction

Software vulnerability is a serious threat to security (Li et al., 2018; Bo et al., 2023). Effective vulnerability detection is crucial for vulnerability repair and thus helpful for mitigating the security threat. Over the years, many source-based methods have been proposed, and they have proven effective for vulnerability detection (Wu and Zou, 2022; Nie et al., 2023; Lin et al., 2019; Wang et al., 2020; Cao et al., 2022). Unfortunately, these methods have limited applicability, as in many cases the program source code is not available. For example, commercial applications and third-party libraries are often closed-source. To cope with this situation, researchers have proposed Binary Software Vulnerability Detection (BSVD) methods (Padmanabhuni and Tan, 2015; Boudjema et al., 2020), among which, static analysis methods are widely used because of their fast detection speed and high usability (Lee et al., 2019; Tian et al., 2020).

Current binary vulnerability detection can be broadly categorized into two types, code similarity-based and pattern-based (Wang et al., 2023a). Code similarity-based methods identify potentially vulnerable code by evaluating the similarity between the code and known vulnerable code in the database (Eschweiler et al., 2016; Feng et al., 2016; Pewny et al., 2015). These methods can often fail to detect vulnerabilities that are semantically similar but syntactically different (Wang et al., 2023b), known as the *recurring* vulnerability. According to the historical data of Open Web Application Security Project (OWASP), the percentage of recurring vulnerabilities (e.g., injection and XSS vulnerabilities) is about 40%, and these vulnerabilities can create (OWASP, 2024) severe security threats.

To be able to also detect the common recurring vulnerabilities, which will also be addressed in this paper, a number of researchers have proposed pattern-based methods that use deep learning models to extract the patterns in vulnerabilities (Lee et al., 2019; Tian et al., 2020). Some of them focus on addressing one or a few specific types of vulnerabilities, such as BVDetector (Tian et al., 2020) and BinVulDet (Wang et al., 2023a). These methods construct the Program Dependency Graph (PDG) and leverage the dependencies to find vulnerabilities related to library/API function calls (e.g., stack-based buffer overflow, heap-based buffer overflow, and integer underflow). Unfortunately, only 46.2% of the vulnerability types in the CWE (Common Weakness Enumeration) catalog are related to calls to library/API

☆ Editor: Dario Di Nucci.
* Corresponding authors.
E-mail addresses: xingchenliang@whu.edu.cn (C. Xing), xxie@whu.edu.cn (X. Xie), qxin@whu.edu.cn (Q. Xin), chengongcg@whu.edu.cn (G. Chen).

https://doi.org/10.1016/j.jss.2025.112658
Received 11 January 2025; Received in revised form 6 September 2025; Accepted 9 October 2025
Available online 14 October 2025

functions (CWE, 2024). This implies the limited applicability of these methods for vulnerability detection.

There are other methods aiming for general vulnerability detection. They are not designed to handle specific types of vulnerabilities. State-of-the-art methods along this line are Instruction2vec (Lee et al., 2019) and HAN-BSVD (Yan et al., 2021). Instruction2vec achieves static binary vulnerability detection by vectorizing assembly instructions using a technique similar to Word2Vec that considers assembly syntax and organizes instructions according to the execution flow (Lee et al., 2019). HAN-BSVD introduces a hierarchical attention network designed to extract vulnerability features from assembly instructions (Yan et al., 2021).

However, the methods for general vulnerability detection have 3 key limitations. **First, they do not leverage high-level code semantic information for vulnerability detection.** These methods extract information from assembly code disassembled from binary code. However, assembly code lacks the abstractions of high-level languages to represent complex logic and data structures, making its semantics limited and difficult to express high-level program intentions (Yang et al., 2023). This results in a lot of vulnerabilities based on complex program logic being difficult to detect, such as memory reuse after free (CWE416) and improper management of system resources (CWE399). On the other hand, the binary code can change considerably due to its sensitivity of the completion configuration and environment (Wang et al., 2023a). The compilation diversity can affect the variation of instructions, making it difficult to extract more robust features for vulnerability detection.

Second, current methods do not make effective use of code structure information for vulnerability detection. Proposed methods such as HAN-BSVD focus only on the representation of the code sequence itself. This means that they would have difficulty representing information about the code's syntactic structure because the sequence of code does not capture code syntactic structure well. Even though a few other methods consider the code structure (Boudjema et al., 2020; Yan et al., 2021; Wang et al., 2023a), they used structure information merely as an aid for obtaining program slices or further acquiring other information, rather than fully utilizing the structural information itself. For example, BinVulDet (Wang et al., 2023a) uses structure information by sequentially examining each node of the AST to determine at which node slicing should be performed. If only a part of the structural information is examined at a time without analyzing the relationships between the different parts, it will be difficult to detect vulnerabilities related to the program's structure effectively. According to the National Vulnerability Database (NVD), 20.22% (7940 in 39274) of vulnerabilities in code are related to code structure (NVD, 2024).

Third, the deep learning models they use have limited generalizability. Proposed methods often use common deep learning models (e.g., TextCNN, BiLSTM-attention network, and hierarchical attention network) that need to learn code features from scratch, lacking general knowledge and strong transferability. They are weak at recognizing complex code and generalizing across datasets with an overfitting tendency.

To address these three limitations, we use multiple code representations to extract information from binary code and use pre-trained models to improve generalization. Specifically, to address the first limitation, we introduce pseudo-code instead of assembly code because pseudo-code is a higher-level abstraction and has more high-level semantic information than assembly code. Also, pseudo-code is not highly sensitive to the diversity of compilation and is reliable for robust feature extraction, as we will further illustrate in Section 2.3. To address the second limitation regarding the lack of code structure information, we use Abstract Syntax Trees (ASTs), which contain rich information about static code structures for vulnerability detection. To address the third limitation, we introduce the pre-trained model GraphCodeBERT for code embedding and vulnerability classification. Traditional deep learning models are typically trained on limited datasets. In contrast, GraphCodeBERT acquires broad and generalizable knowledge due to two reasons. First, GraphCodeBERT's model pre-training is performed on the large CodeSearchNet (Husain et al., 2019) dataset, encompassing approximately 6 million functions from open-source code across multiple programming languages. Second, the model accounts for code's intrinsic structure, using both the AST for syntax and the Data Flow Graph (DFG) for semantic representation. This general knowledge helps GraphCodeBERT adapt to a wider variety of data types, thereby enhancing our model's generalization ability.

We propose *PreMulBVD*, a pretraining-based binary vulnerability detection method using the AST and pseudo-code. *PreMulBVD* has four main steps. In the first step, it decompiles binary code into pseudo-code to model the code's sequential structure, while simultaneously extracting ASTs from the pseudo-code to model its syntactic structure. *PreMulBVD* uses structured pseudo-code containing structured programming elements such as conditional statements, loop statements, etc. to describe algorithms and program logic. In the second step, *PreMulBVD* processes the pseudo-code and ASTs via for example code normalization and construction of an AST in a graph form, ensuring that the input format meets the requirements of the model. Next, in the third step, we fine-tune the pre-trained model using processed AST and pseudo-code as training data, enhancing its adaptability to vulnerability detection tasks. Finally, it uses the fine-tuned model to predict whether the code to be tested contains vulnerabilities.

To evaluate the effectiveness of *PreMulBVD*, we conducted experiments on a dataset containing 119 various CWE vulnerability types and compared *PreMulBVD* with three SOTA pattern-based methods for general vulnerability detection. Our results show that *PreMulBVD* achieve over 97% accuracy, precision, recall, and f1-score and it outperforms SOTA methods with higher accuracy (3.34%) and f1-score (4.40%). These results demonstrate *PreMulBVD*'s superior ability for general vulnerability detection.

The main contributions of this paper can be summarized as follows:

(1) We propose *PreMulBVD*, a novel method for binary code vulnerability detection that addresses three key limitations of existing methods by utilizing ASTs and pseudo-code to represent the sequential semantics and structural information of the code, and is the first to leverage pre-trained models designed for source code to extract binary code vulnerability features.
(2) We implement and open-source the prototype of *PreMulBVD*.[1] To validate the detection capability of *PreMulBVD* for various types of vulnerabilities, we have organized a dataset containing 119 CWE categories, nearly covering all the datasets used in existing work.
(3) We designed and implemented a series of experiments to validate how well *PreMulBVD* compares to other baseline methods, as well as to validate the impact of the different components of *PreMulBVD* on vulnerability detection capabilities.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to background knowledge. The presented method is introduced in more detail in Section 3. Experimental setup and result analysis are described in Sections 4 and 5, respectively. We will discuss potential threats in Section 6. Section 7 summarizes the research background and related work. In Section 9 we summarize the full text.

2. Background

2.1. Pattern-based method vs similarity-based method

Modern static binary code vulnerability detection techniques can generally be divided into two categories: those identifying vulnerabilities by comparing code under test with known vulnerable code

[1] https://github.com/galaxyview/PreMulBVD

(similarity-based) and those detecting vulnerabilities by extracted vulnerability patterns (pattern-based).

Code similarity-based methods entail building a database of known vulnerable code snippets and then evaluating the similarity between the target code and these snippets. The target code is flagged as potentially vulnerable when the similarity exceeds a specific threshold. The most straightforward approach to measure the similarity of binary functions is by using assembly code content to compute edit distances and detect function similarities. However, code similarity-based methods have a significant drawback in that they can only identify known vulnerabilities (Yang et al., 2021, 2023; Luo et al., 2023), making them difficult to recognize recurring vulnerabilities that share similar code logic with previously identified vulnerabilities.

To tackle recurring vulnerabilities, some researchers have proposed pattern-based methods that leverage deep learning techniques to identify patterns or signatures signaling vulnerabilities in the target code (Lee et al., 2019; Tian et al., 2020; Yan et al., 2021; Wang et al., 2023a). Specifically, pattern-based approaches represent binary programs as vectors or graphs and extract patterns of vulnerabilities from these vectors using deep learning methods, after which the binary code to be tested is simply represented as vectors and then input to the model to determine whether a vulnerability exists.

To effectively address recurring vulnerabilities, *PreMulBVD* proposed in this paper is a pattern-based method.

2.2. Pre-trained model

Large-scale pre-trained models like BERT (Devlin, 2018) and GPT (Radford, 2018) have recently achieved significant success. They have been effectively applied to a variety of downstream code-related tasks, such as code search, clone detection, code translation, and code refinement (Feng et al., 2020). On this basis, pre-trained models based on the BERT architecture and targeting code-related tasks have been widely proposed. Feng et al. introduced CodeBERT (Feng et al., 2020), a bimodal pre-trained encoder-only model. It was trained on 2 million functions from six programming languages, including Java and Python, and on natural language documentation like code comments. Although CodeBERT has proven effective for tasks related to vulnerabilities (Hin et al., 2022), it treats source code as a sequence of tokens similar to natural language, overlooking the structural information inherent in the code.

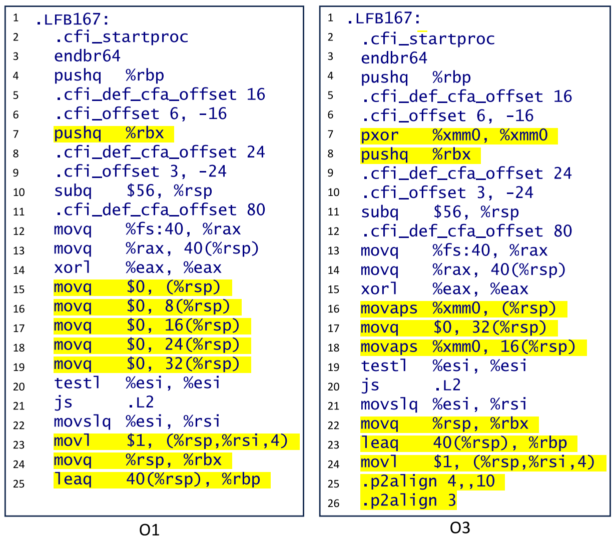
Since CodeBERT is unable to efficiently process the structural information of the code, which leads to its insufficient understanding of the code's semantics, ASTs have long been used to extract the semantic information of the code, in order to complement the pre-trained model's ability to this regard, Guo et al. proposed GraphCodeBERT (Guo et al., 2020), which builds upon the same architecture as CodeBERT but includes data flow analysis. It introduces two structure-aware pretraining tasks — data flow edge prediction and node alignment — to represent source code and data flow better, enhancing CodeBERT's capabilities.

There is already work on vulnerability detection by applying pre-trained models to source code representations (Wang et al., 2024; Liu et al., 2024; Zhang et al., 2024). However, to the best of our knowledge, there is no work on applying them to the field of binary vulnerability detection, probably because the vast majority of pre-trained models are trained based on source code, and it is not possible to apply them directly to a representation of binary code.

PreMulBVD innovatively further processes assembly code into pseudo-code so that it can be understood by pre-trained models.

2.3. Impact of compilation diversity

Most binary vulnerability detection methods extract features after decompiling binary code into assembly code. However, assembly code can be very different due to the diversity of compilation. In practical, vulnerabilities usually originate from small portions of instructions in



```
.LFB167:
  .cfi_startproc
  endbr64
  pushq   %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  pushq   %rbx
  .cfi_def_cfa_offset 24
  .cfi_offset 3, -24
  subq    $56, %rsp
  .cfi_def_cfa_offset 80
  movq    %fs:40, %rax
  movq    %rax, 40(%rsp)
  xorl    %eax, %eax
  movq    $0, (%rsp)
  movq    $0, 8(%rsp)
  movq    $0, 16(%rsp)
  movq    $0, 24(%rsp)
  movq    $0, 32(%rsp)
  testl   %esi, %esi
  js      .L2
  movslq  %esi, %rsi
  movl    $1, (%rsp,%rsi,4)
  movq    %rsp, %rbx
  leaq    40(%rsp), %rbp
```

O1

```
.LFB167:
  .cfi_startproc
  endbr64
  pushq   %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset 6, -16
  pxor    %xmm0, %xmm0
  pushq   %rbx
  .cfi_def_cfa_offset 24
  .cfi_offset 3, -24
  subq    $56, %rsp
  .cfi_def_cfa_offset 80
  movq    %fs:40, %rax
  movq    %rax, 40(%rsp)
  xorl    %eax, %eax
  movaps  %xmm0, (%rsp)
  movq    $0, 32(%rsp)
  movaps  %xmm0, 16(%rsp)
  testl   %esi, %esi
  js      .L2
  movslq  %esi, %rsi
  movq    %rsp, %rbx
  leaq    40(%rsp), %rbp
  movl    $1, (%rsp,%rsi,4)
  .p2align 4,,10
  .p2align 3
```

O3

Fig. 1. Assembly code generated by different optimization levels.

a program (Wang et al., 2023a), which means that the variability introduced by different compilations can change these instructions, thus weakening the robustness of the features extracted from assembly code.

This is so common that whenever the compilation environments are not exactly the same, the generated assembly code may differ. Here is an example that supports this idea. We use the state-of-the-art decompilation tool IDA Pro (hex rays, 2019) to decompile binary code, generated by compiling the same source code with different compiler optimization levels, into pseudo-code and assembly code. Fig. 1 illustrates assembly code compiled from the same program at different optimization levels, and it is easy to see that the two pieces of code are very different at two optimization levels, O1 and O3. In contrast, Fig. 2 shows further pseudo-code generated from assembly code for the same program at different optimization levels, and it can be found that the difference between these two pieces of pseudo-code is significantly smaller than that of assembly code, except for the naming of some variables and the way they are declared, which has less impact on the understanding of the program's semantics than the difference in the assembly code. In addition, we randomly selected 100 functions and compiled them into binary code using optimization levels O1 and O3. These binary codes were then decompiled into assembly code and pseudo-code, respectively. We compared the proportion of different statements in the two code segments under different compilation options to the total number of statements. The result showed that the proportion for assembly code was 37.3%, while for pseudo-code, it was 17.6%. The proportion for pseudo-code was 19.7% lower than that for assembly code.

This suggests that pseudo-code can effectively mitigate the variations in assembly code due to different compilation environments, thus reducing the impact on the robustness of vulnerability detection.

3. Methodology

The system architecture of *PreMulBVD* is shown in Fig. 3. *PreMulBVD* operates in four steps: (1) decompiling binary code to pseudo-code; (2) parsing ASTs and pseudo-code; (3) training a prediction model for vulnerability detections; and (4) using the trained model to detect vulnerability.

Initially, binary programs are decompiled into assembly code, which is further processed to generate pseudo-code, including steps such as building control flow diagrams, data flow analysis, and optimization of the code. Then, we process the pseudo-code by normalizing it,

```
unsigned __int64 __fastcall CWE121_...::CWE121_...(
        CWE121_...::CWE121_... *this,int a2)
{
  __int64 *v2; // rbx
  __int64 v4[5]; // [rsp+0h] [rbp-48h] BYREF
  unsigned __int64 v5; // [rsp+28h] [rbp-20h] BYREF

  v5 = __readfsqword(0x28u);
  memset(v4, 0, sizeof(v4));
  if ( a2 < 0 )
  {
    printLine("ERROR: Array index is negative.");
  }
  else
  {
    *((_DWORD *)v4 + a2) = 1;
    v2 = v4;
    do
    {
      printIntLine(*(unsigned int *)v2);
      v2 = (__int64 *)((char *)v2 + 4);
    }
    while ( v2 != (__int64 *)&v5 );
  }
  return __readfsqword(0x28u) ^ v5;
}
```

O1

```
unsigned __int64 __fastcall CWE121_...::CWE121_...(
        CWE121_...::CWE121_... *this,int a2)
{
  __int64 *v2; // rbx
  __int64 v3; // rdi
  __int128 v5[2]; // [rsp+0h] [rbp-48h] BYREF
  __int64 v6; // [rsp+20h] [rbp-28h]
  unsigned __int64 v7; // [rsp+28h] [rbp-20h] BYREF

  v7 = __readfsqword(0x28u);
  memset(v5, 0, sizeof(v5));
  v6 = 0LL;
  if ( a2 < 0 )
    return printLine("ERROR: Array index is negative.");
  v2 = (__int64 *)v5;
  *((_DWORD *)v5 + a2) = 1;
  do
  {
    v3 = *(unsigned int *)v2;
    v2 = (__int64 *)((char *)v2 + 4);
    printIntLine(v3);
  }
  while ( v2 != (__int64 *)&v7 );
  return __readfsqword(0x28u) ^ v7;
}
```

O3

Fig. 2. Pseudo-code generated by different optimization levels.

including steps such as unifying lengths and renaming variable names. The ASTs are extracted from the pseudo-code and transformed into a graph. Next, the pre-trained language model encodes pseudo-code and ASTs, and the model is fine-tuned on our dataset for predicting vulnerabilities. Finally, we use the trained model to determine whether the binary code under test contains vulnerabilities.

3.1. Decompiling

Vulnerability detection tools for binary programs usually disassemble the binary code into assembly code (Lee et al., 2019; Yan et al., 2021; Tian et al., 2020). However, there is an obvious limitation to using assembly code for vulnerability detection as we mentioned in Sections 1 and 2.3, i.e., lack of high-level semantic information and robustness.

To address the limitation, we further generate pseudo-code from assembly code to minimize the impact of assembly code diversity on the robustness of the feature extraction, while providing more high-level semantic features. Fig. 4 shows an example of generating pseudo-code from assembly code. The steps that *PreMulBVD* takes to get the pseudo-code are as follows: first, *PreMulBVD* uses disassemble tools to disassemble the target binary file, converting the binary code into assembly language code. After disassembling the binary file, the decompiler analyzes the program's control flow by identifying function entry points, branches, jump instructions, etc. It constructs the control flow graph (CFG) to help identify the logical structure of the program. The assembly code typically contains symbols like memory addresses and registers. Decompilers assign default names to symbols, typically based on variable types and locations, function call relationships, memory layouts, data flows, and contextual reasoning to automatically generate symbol names. As shown in Fig. 4, the naming order of `v1` and `v2` in the pseudo-code is derived from their positions on the stack, which are `[rbp-8]` and `[rbp-4]`, respectively. After control flow analysis and symbol renaming, the decompiler converts the assembly instructions into pseudo-code by first parsing the instruction and mapping it to C-like operations, such as assignment statements, arithmetic operations, and conditional statements. For example, `mov eax, 0` would be translated to `eax = 0`. It infers the type of operands, such as pointers, integers, or constants, by analyzing the operands of assembly instructions. Additionally, it determines the control flow structure, such as conditional branches and loops, by examining the jump relationships between instructions. By combining data flow analysis, control flow analysis, and type inference, the decompiler generates readable pseudo-code in a high-level language style. After generating a rough pseudo-code, the decompiler automatically corrects some issues by analyzing the program's control flow graph and data flow to ensure the logical correctness of the pseudo-code. It addresses potential problems caused by deviations or irregularities in the disassembly process, such as automatically adjusting the control flow, fixing invalid jumps, or correcting the instruction order.

It is worth noting that in *PreMulBVD*, we use IDA Pro as the decompiler. However, there are also other advanced decompilers, such as Ghidra. Therefore, to analyze the impact of decompilers on the vulnerability localization capability of *PreMulBVD*, we will further explore this issue in RQ4.

After completing these steps, we can get the pseudo-code for the assembly code.

3.2. Preprocessing and parsing

In this step, we need to process and parse the pseudo-code obtained in the decompiling step. Specifically, we need to remove some identifiers, comments, and constants from the pseudo-code to eliminate their impact on vulnerability detection. We also need to build ASTs from pseudo-code to represent code structure information.

3.2.1. Normalizing pseudo-code

It is important to note that the pseudo-code also requires some processing to remove some elements that interfere with vulnerability detection. Pseudo-code generated by decompilers often includes identifiers, constants, and comments that are not related to the functionality of the code. They interfere with the model's learning of patterns, which in turn negatively affects the results of vulnerability detection. Therefore, we normalize the pseudo-code before using it. Specifically, normalization is carried out with the following operations:

- Unify identifier names to include all variable names and function names for non-external function calls. For example, variable names and function names are unified to VAR_NAME and FUNC_NAME, respectively.
- For all types of specific values, all are uniformly replaced with placeholders corresponding to the value type. For example, string values and number values are unified to STRING_CONST and NUM_CONST, respectively.
- Remove all automatically generated comments.

After this step, *PreMulBVD* gets the normalized pseudo-code.

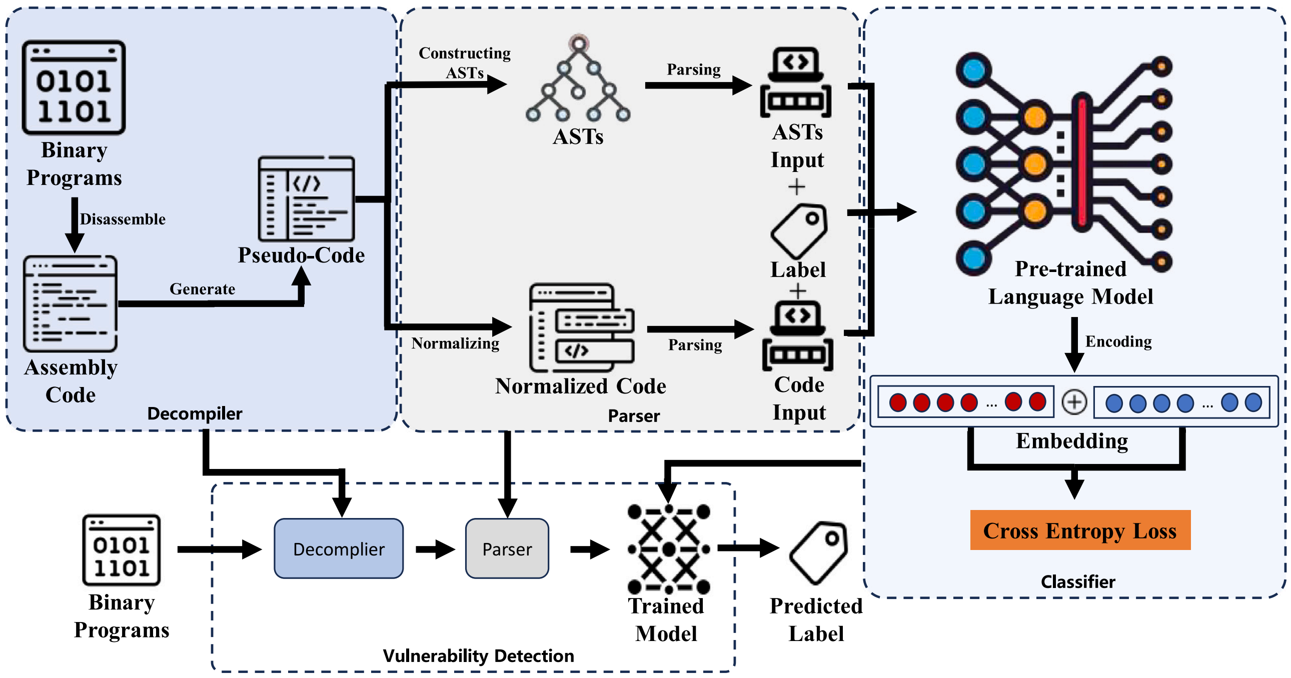
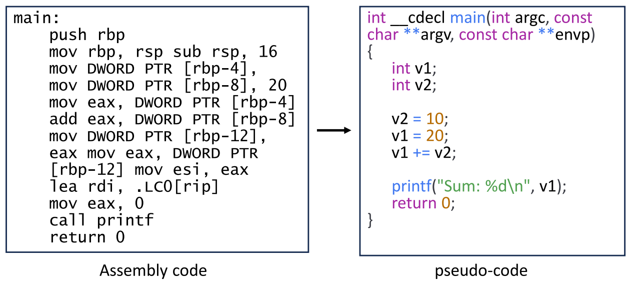


Fig. 3. The overview of *PreMulBVD*. *PreMulBVD* consists of four steps. ❶ In the decompiling step (Section 3.1), binary programs are decompiled into assembly code, which is further processed to generate pseudo-code. ❷ In the parsing step (Section 3.2), we parse the information in the pseudo-code by normalizing it, and the ASTs are extracted from the pseudo-code and transformed into a graph. ❸ In the model constructing step (Section 3.3), the pre-trained language model encodes pseudo-code and ASTs, and the pre-trained model is fine-tuned on our dataset to make it competent for predicting vulnerabilities. ❹ In the vulnerability detection step (Section 3.4), we use the trained model to classify binary code to be tested for vulnerability detection.



```
main:
    push rbp
    mov rbp, rsp sub rsp, 16
    mov DWORD PTR [rbp-4],
    mov DWORD PTR [rbp-8], 20
    mov eax, DWORD PTR [rbp-4]
    add eax, DWORD PTR [rbp-8]
    mov DWORD PTR [rbp-12],
    eax mov eax, DWORD PTR
    [rbp-12] mov esi, eax
    lea rdi, .LC0[rip]
    mov eax, 0
    call printf
    return 0
```

Assembly code

```
int __cdecl main(int argc, const
char **argv, const char **envp)
{
    int v1;
    int v2;

    v2 = 10;
    v1 = 20;
    v1 += v2;

    printf("Sum: %d\n", v1);
    return 0;
}
```

pseudo-code

Fig. 4. An example of generating pseudo-code from assembly code.

3.2.2. *Constructing ASTs*

To address the limitation of the lack of syntactic structure information in existing methods, *PreMulBVD* models the syntactic structure of programs and summarizes the vulnerability patterns.

Since the AST directly represents the syntactic structure of a program, vulnerability detection tools can quickly find these vulnerabilities through pattern matching. In fact, many security vulnerabilities can often be recognized by fixed syntax patterns, which can be efficiently extracted by AST. For example, detecting calls to specific functions (e.g., `system()`) or checking the use of certain unsafe library functions (e.g., `gets()` and `scanf()`) can be accomplished by looking for specific function nodes or calling patterns in the AST.

Compared to ASTs, PDGs focus on the control flow and data flow of a program, which is more complex information. Nodes in PDGs are not just simple syntax elements, but they also involve data dependencies between parts of the program. For example, in PDG, it is necessary to analyze how the flow goes from the assignment of one variable to another, or how to control the execution path of the code through the branches of the program. This makes pattern matching in PDG more complicated and leads to poor robustness of the final extracted patterns. AST also serves as a foundation for the construction of the Control Flow Graph (CFG) and Data Flow Graph (DFG). It preserves the syntactic hierarchy of a program and distinguishes the various code components by their syntactic structures including the conditional branches, loop structures, variable declarations, and the nesting relationships of function calls. Unlike AST, CFG focuses on the execution paths of code blocks and DFG captures the data flow. They do not preserve enough information about the program's syntactic structure. For example, CFG does not capture the internal structure of a basic block. As explained in the paper, PreMulBVD aims to make effective use of the syntactic structure of the program to improve vulnerability detection, and it uses the AST, which encodes more information about the syntactic structure of the program. Therefore, considering the trade-off between effectiveness and efficiency, *PreMulBVD* chooses AST rather than PDG, CFG or DFG to model the syntax structure.

We use IDA Pro to extract ASTs from pseudo-code. Extracting ASTs from pseudo-code usually involves several steps, starting with converting pseudo-code into tokens, the smallest meaningful units such as keywords, identifiers, constants, and operators. Meanwhile, the AST generator infers variable values through static analysis, relying on techniques like control flow analysis, data flow analysis, and constant propagation. The AST generator examines the program's structure and instruction sequences to deduce how values are propagated across variables without requiring the program to be executed. Next, the AST generator converts tokens into an AST based on the language's syntax rules. Then, the AST generator validates the AST for semantic correctness, checking types, scopes, etc. Finally, the AST generator optimizes and generates the AST, such as removing invalid code, merging constant expressions, etc.

In fact, there are a few models that can directly accept tree-structured ASTs, and these models are usually designed specifically for processing tree-structured or graph-structured data, such as Graph Neural Networks (GNN) and Tree-LSTM. However, many advanced deep learning models, such as Transformer, do not directly process tree structures and they need first to convert ASTs to sequence or graph structures. Therefore, to utilize these advanced deep learning models, we chose to further construct the AST as a graph structure.

Table 1
Statements and expressions in ASTs.

#	Node type	Description
Statements	if	if statement
	block	instructions executed sequentially
	for	for loop statement
	while	while loop statement
	switch	switch statement
	return	return statement
	goto	unconditional jump
	continue	continue statement in a loop
	break	break statement in a loop
Expression	assignment	assignment, assignment after or, xor, and, add, sub, mul, div
	comparisons	equal, not equal, greater than, less than, greater than or equal to, and less than or equal to
	arithmetic	or, xor, addition, subtraction, multiplication, division, not, post-increase, post-decrease, pre-increase, and pre-decrease
	other	indexing, variable, number, function call, string, asm, and so on

The overview of generating the AST graph is shown in Fig. 5. To convert an AST to a graph, each AST node is kept as a graph node, parent–child relationships become directed edges, and sibling nodes are connected with additional edges to enhance connectivity. We define an AST of function f_i as $AST_i = (V_i, E_i)$, where V_i represents a set of nodes, each node represents a pseudo code statement or control predicate, and E_i represents the structure relationship between each node. Each $node_i \in V_i = \langle id_i, opname_i, value_i \rangle$, where id_i is an identifier that is unique to the $node_i$, $value_i$ is the value of the variable if $node_i$ represents variable nodes, which was previously saved in the AST node, and $opname_i$ is the name of the operation of the pseudo-code corresponding to the $node_i$. We classify $opname$ in an AST into two categories based on their functionalities as shown in Table 1: (i) statement nodes and (ii) expression nodes. Each $edge_i \in E_i = \langle source_i, target_i \rangle$, where $source_i$ is the id of the source node and $target_i$ is the id of the target node. At the end of this step, we get the AST in the form of a graph extracted from the pseudo-code.

3.3. Constructing model

Previous methods of generating embeddings for assembly code or pseudo-code often involved training static word embedding models, such as Word2vec (Mikolov, 2013; Yan et al., 2021), on a dataset of code snippets to produce vectors for each code token. As for generating embeddings of tree-structured inputs, previous methods often involve models that maximize the retention of tree-structured features, such as Graph Neural Network (GNN) (Li et al., 2023) and Tree-LSTM (Yang et al., 2021, 2023). Encoding code or AST with traditional models often struggles with capturing global dependencies, understanding complex semantics, and effectively modeling cross-scope relationships (Liu et al., 2024). These models are typically limited to local structural features, lack flexibility for multimodal inputs, and require extensive customization for different tasks or languages.

To address these drawbacks, *PreMulBVD* leverages the power of large-scale pre-trained code language models, utilizing extensive pre-training to acquire prior knowledge for initializing code and graph embeddings. Using pre-trained models will solve the problems of insufficient contextual understanding and comprehension of complex semantic information in traditional models. We implement the node representations by invoking the pre-trained code language model on the target dataset. Specifically, we use the pre-trained programming language model GraphCodeBERT (Guo et al., 2020) to initialize the embedding of ASTs and pseudo-code fragments. ASTs and pseudo-code fragments processed by GraphCodeBERT will output two high-dimensional word embedding vectors. To enable the pre-trained model to perform our prediction task, we designed a Multi-Layer Perceptron (MLP) to classify the data. The input layer takes as input the code and ASTs embedding vectors obtained from GraphCodeBERT and outputs the classification result after passing through two fully connected layers.

The first layer projects the input feature vector from 768 dimensions to 128 dimensions. This layer automatically learns a weight matrix and a bias vector (Eq. (1)), where W is the learned weight matrix, x is the input feature vector, b is the bias, and y is the output of the linear layer, which is updated by backpropagation.

$$y = W \cdot x + b \tag{1}$$

After that, the activation function (ReLU(.)) performs a nonlinear transformation of the 128-dimensional vector, which is used as the activation function to increase sparseness. The ReLU(.) function is shown in Eq. (2), which nonlinearly transforms the output of the linear layer.

$$ReLU(x) = max(0, x) \tag{2}$$

Next, the second linear layer maps the ReLU-activated 128-dimensional features to 1 dimension for outputting a scalar. Then, the Sigmoid function (Eq. (3)) converts the output value of the linear layer into a probability value ranging from 0 to 1. The value measures how vulnerable the code is.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

For binary classification tasks, the Binary Cross-Entropy (BCE) is commonly used to measure the accuracy of model predictions (Zhang and Sabuncu, 2018). As a result, we use BCE to fine-tune our model, which is represented as follows:

$$\mathcal{L}(y, \hat{y}) = -y \cdot log(\hat{y}) - (1 - y) \cdot log(1 - \hat{y}) \tag{4}$$

where y is the true label (0 or 1) and $\hat{y}$ is the probability of belonging to category 1 as predicted by the model.

The AdamW optimizer is commonly used for GraphCodeBERT fine-tuning. AdamW optimizer is a variant of the Adam optimizer that incorporates Weight Decay to effectively prevent model overfitting (Loshchilov et al., 2017). The update formula for the AdamW optimizer is:

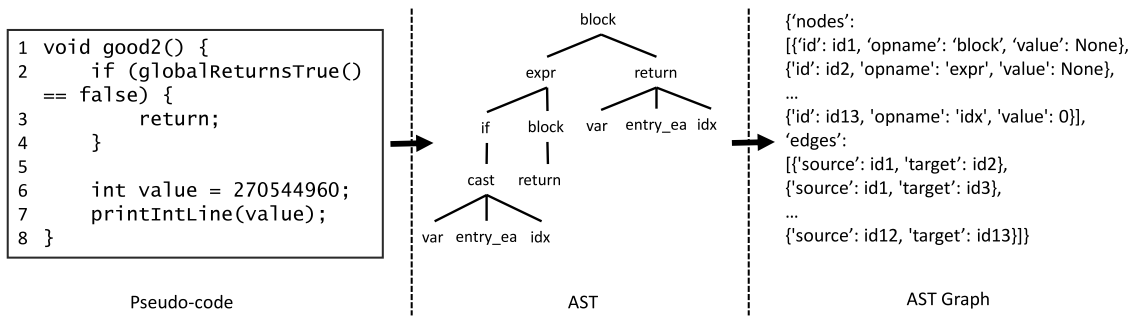
$$\theta_t = \theta_{t-1} - \eta \cdot \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda\theta_{t-1}\right) \tag{5}$$

where η is the learning rate, $\hat{m}_t$ is the first-order momentum, $\hat{v}_t$ is the second-order momentum, and λ is the weight decay coefficient.

Here is the fine-tuning process. We first initialize the self-embedding layer using the pre-trained embeddings from GraphCodeBERT. Simultaneously, we fine-tuned the model on the dataset to enhance the model's ability for vulnerability detection, thus improving the accuracy of the prediction results. Finally, we obtain a deep learning model capable of vulnerability detection.

3.4. Detecting vulnerability

Finally, *PreMulBVD* uses the trained model to detect vulnerabilities that exist in the program under test. More specifically, *PreMulBVD* feeds the program under test to the trained model, and the decompiler converts it into pseudo-code. Then the parser normalizes pseudo-code and constructs ASTs. Next, the trained model will be used to process it and finally output the prediction results. The predictions given by the system can help developers localize the problematic vulnerabilities for fixing.



```
void good2() {
    if (globalReturnsTrue()
== false) {
        return;
    }

    int value = 270544960;
    printIntLine(value);
}
```

```
{'nodes':
[{'id': id1, 'opname': 'block', 'value': None},
{'id': id2, 'opname': 'expr', 'value': None},
...
{'id': id13, 'opname': 'idx', 'value': 0}],
'edges':
[{'source': id1, 'target': id2},
{'source': id1, 'target': id3},
...
{'source': id12, 'target': id13}]}
```

Fig. 5. The steps of AST graph generation.

4. Experiment setup

In this section, we describe the experiment setup including the datasets, research questions, evaluation metrics, and the baselines.

4.1. Research questions

We have formulated the following 6 Research Questions (RQs):

- **RQ1: How does the pattern-based *PreMulBVD* compare to state-of-the-art pattern-based methods in terms of vulnerability detection?**
  To evaluate the performance of *PreMulBVD*, we empirically compare its effectiveness (i.e., accuracy and f1-scores) with state-of-the-art (SOTA) generally applicable pattern-based vulnerability detection methods.
- **RQ2: How effective are the different components of *PreMulBVD*?**
  We designed this research problem to explore the impact of different information extracted from binary code on the effectiveness of vulnerability detection, respectively.
- **RQ3: How do different pre-trained models influence the overall performance of *PreMulBVD*?**
  In this paper, we use GraphCodeBERT for code and graph embedding by default. However, there are many similar pre-trained models. We use this research question to investigate the influence of different models on code vulnerability detection performance.
- **RQ4: How do pseudo-code and ASTs generated by different decompilers impact vulnerability detection capabilities of *PreMulBVD*?**
  We default to using the advanced decompiler IDA Pro (hex rays, 2019). However, to study the impact of different decompilers on the generated pseudo-code and ASTs regarding *PreMulBVD* vulnerability detection capabilities, we will utilize RQ5 for investigation.
- **RQ5: How much improvement in vulnerability detection capability is achieved by using pseudo-code compared to directly using assembly code?**
  In this paper, we use pseudo-code instead of assembly code to represent binary code. In this research question, we will investigate to what extent using pseudo-code improves vulnerability detection capability compared to using assembly code.
- **RQ6: How much does *PreMulBVD* vary in its ability to detect vulnerabilities in binary code generated at different optimization levels?**
  We consistently use the default optimization levels to compile the source code in the Juliet Test Suite and generate binary code. In this research question, we will investigate whether there is a difference in the ability of *PreMulBVD* to detect vulnerabilities in binary code generated at different optimization levels.

4.2. Datasets preparation

To cover as many types of vulnerabilities as possible, we do not want to directly use the datasets used in the proposed methods, as they often contain only a few types of vulnerability. For example, the authors of HAN-BSVD and instruction2vec select the Stack-based Buffer Overflow category, and BVDetector selects the Memory Corruption and Number Handling categories. Instead, we used the dataset Juliet Test Suite v1.3 for C/C++ SARD, which covers almost all the vulnerability categories contained in the datasets used in the proposed methods. The dataset contains a large number of production, synthetic, and academic programs divided into 119 CWE directories. Each of the categories contains a varying number of programs that are classified as either “bad” (vulnerable) or “good” (non-vulnerable, which represents a vulnerable patched version of programs). The dataset is provided with the source code of the program, so we need to compile it to make it into binary code, using the following steps:

(1) Data organization: Each vulnerable program in the Juliet test suite has at least one CWE (Common Weakness Enumeration) ID, which represents different types of vulnerabilities. We selected all 119 CWE categories. Due to the large total amount of data, too much data may result in long compilation, training, and testing times. To cover as many vulnerability types as possible, we randomly selected 20% of the functions of each category in the Juliet Test Suite as the final data set.

(2) Compile: We chose source code that can compile correctly based on a Linux environment and compiled it using the Makefile file that comes with the Juliet test suite, using the most widely used compiler, GCC, and the default optimization levels. In the end, we used a total of 126,142 binaries containing 101,798 non-vulnerable functions and 24,344 functions containing vulnerabilities. In each experiment, we randomly split the training set, validation set, and test set in a ratio of 8:1:1.

(3) Disassembly: In order to extract pseudo-code and ASTs from binary programs, we used IDA Pro 8.3 (the most popular binary code decompilation tool, which excels at the task of recovering binary code (hex rays, 2019)) to perform disassembly operations.

(4) Normalization: After extracting the pseudo-code, we normalized it according to the rules mentioned in Section 3.1. To accurately identify the components of the pseudo-code, we use the popular parser generator tool Tree-sitter to analyze and normalize each part accordingly.

4.3. Evaluation metrics

TP, TN, FP, and **FN** are the numbers of true positive results, true negative results, false positive results, and false negative results, respectively.

We used four commonly used indicators, Precision, Recall, F1-Score, and Accuracy, calculated as follows:

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

$$F1 - Score = 2 \times \frac{TP}{2TP + FP + FN} \tag{9}$$

4.4. Baselines

In RQ1, in order to evaluate the effectiveness and efficiency of *PreMulBVD*, we empirically compare *PreMulBVD* and state-of-the-art pattern-based binary code vulnerability detection methods attempting to address all types of vulnerabilities. Below are summaries of the three techniques:

- **HAN-BSVD** (Yan et al., 2021): HAN-BSVD enhances binary software vulnerability detection by enriching contextual information through preprocessing, embedding instructions with Bi-GRU and attention, and extracting features with TextCNN and spatial attention to highlight crucial regions. To the best of our knowledge, it is the **state-of-the-art** binary vulnerability detection method that is not restricted to certain types of vulnerabilities. *(Note that since the authors did not make the code public, we reproduced it by ourselves.)*
- **Instruction2vec** (Lee et al., 2019): Instruction2vec has designed a word vectorization model for the characteristics of assembly instructions, using a technique similar to Word2Vec to vectorize the assembly instructions, allowing for the assembly syntax, and organize the instructions according to the execution flow. Then it uses Text-CNN to learn and extract features from assembly code automatically. Instruction2vec effectively enhances the detection of static binary vulnerabilities. *(Note that we used the code provided by the authors for our experiments.)*
- **VulPin** (Chukkol et al., 2023): VulPin uses fine-grained slicing and a hierarchical attention network (HAN) to analyze binary code, isolating vulnerability-related sections and filtering out noise. It captures semantic data at both slice and basic block levels, employing attention mechanisms and residual connections to prioritize influential code segments. *(Note that since the authors did not make the code public, we reproduced it by ourselves.)*

Unlike HAN-BSVD, which directly uses Word2Vec, Instruction2vec has specially designed the word vectorization model for the characteristics of assembly instructions, so that it can vectorize according to the characteristics of assembly code. In feature learning, unlike Instruction2vec, which directly uses Text-CNN to automatically learn and extract features from assembly code, HAN-BSVD employs a multi-level deep learning model, including hierarchical attention networks and Bi-GRU, to more fully extract features from binary code. VulPin captures semantic information at the slice and basic block levels, unlike HAN-BSVD and Instruction2vec, which operate at the sequence level. Therefore, to explore the advantages of *PreMulBVD* in binary code representation and feature learning, we selected these three as baseline methods for comparison.

In RQ3, we would like to compare the impact of different pretraining models on vulnerability detection, in order to include the most advanced and classical pretraining models for code representation as much as possible, we pick the most classical code representation model CodeBERT (Guo et al., 2020), and two models that advance CodeBERT, namely GraphCodeBERT (Guo et al., 2020) and ContraBERT (Liu et al., 2023). Additionally, we have incorporated UniXcoder (Guo et al., 2022), a pre-trained model focused on system-level programming and command-line interfaces, to validate models of types other than CodeBERT-like models. Below are the summaries of these models:

Table 2
Network parameters in our experiments.

Parameter	Value
Batch size	64
Learning rate	0.00001
Epoch	30
Embedding size	768*2
Dropout	0.5

- **GraphCodeBERT** (Guo et al., 2020): Guo et al. extended the Transformer neural architecture by introducing a graph-guided masked attention function to incorporate the code structure and proposed GraphCodeBERT.
- **CodeBERT** (Feng et al., 2020): CodeBERT is a bimodal pretraining model for natural and programming languages based on a multilayer bi-directional Transformer.
- **ContraBERT** (Liu et al., 2023): ContraBERT enhances model robustness using contrastive learning by applying nine data augmentation operators to generate semantically equivalent PL-NL variants. The model is further trained with masked language modeling (MLM) and a contrastive pretraining task on both original and augmented data.
- **UniXcoder** (Guo et al., 2022): UnixCoder is a bimodal pretraining model for system programming and commands based on a multilayer bi-directional Transformer.

It is worth noting that UniXcoder, CodeBERT, and ContraBERT did not undergo specialized training for spatial structure data such as ASTs during their pre-training. Therefore, they treat the input AST sequences as regular sequences for learning. In contrast, GraphCodeBERT was specifically trained on AST structures during its pre-training, resulting in a stronger ability to represent syntactic relationships and code dependencies in ASTs.

In the decompilation phase, we have defaulted to using IDA Pro 8.3 (hex rays, 2019) as the decompiler to generate pseudo-code, as it is the most widely used and advanced. However, to investigate the impact of decompilers on the *PreMulBVD*'s vulnerability detection ability, we have also introduced another advanced decompiler, Ghidra 11.1 (N.S.A., 2021), for comparison in RQ4 and RQ5. Compared to IDA Pro, Ghidra is open-source and free, supports multiple platforms and architectures, and is therefore widely used. Below is an introduction to both decompilers:

- **IDA Pro** (hex rays, 2019): A powerful disassembler and debugger with interactive features, multi-architecture support, and Hex-Rays decompiler for near-source analysis, ideal for reverse engineering.
- **Ghidra** (N.S.A., 2021): An open-source reverse engineering tool by the NSA, offering disassembly, decompilation, debugging, and scripting support, with strong community-driven development.

4.5. Experiment environment

Table 2 shows the experiment environment. Each model is implemented using Pytorch v1.14.0 and trained by a machine with Intel Xeon(R) Gold 6254 CPU, 32 GB RAM, two 48G NVIDIA A6000 GPUs, and Ubuntu 20.04.3 LTS operation system.

5. Results and analysis

In this section, we analyze the results of the experiments for the 6 RQs.

Table 3
Performance of each method on Juliet Test Suite dataset.

Method	Accuracy	F1-Score	Precision	Recall
HAN-BSVD (Yan et al., 2021)	0.9556	0.9321	0.9310	0.9332
Instruction2vec (Lee et al., 2019)	0.9108	0.8910	0.8835	0.8986
VulPin (Chukkol et al., 2023)	0.9489	0.9219	0.9134	0.9320
PreMulBVD	**0.9875**	**0.9731**	**0.9738**	**0.9720**

5.1. RQ1: Comparison of *PreMulBVD* with other methods

To evaluate the effectiveness of *PreMulBVD*, we compared the effectiveness of *PreMulBVD* with other state-of-the-art pattern-based vulnerability detection models on Juliet Test Suite. Since most of the existing work has used Accuracy, F1-score, Precision, and Recall as evaluation metrics (Wang et al., 2023a; Yan et al., 2021; Lee et al., 2019; Xiao et al., 2021), we also adopt them as evaluation metrics in RQ1.

Based on the experimental results in Table 3, we can find that after introducing the pre-trained model to represent pseudo-code and ASTs, *PreMulBVD* outperforms the baseline methods. Compared to the SOTA, i.e., HAN-BSVD, the improvements in terms of Accuracy, F1-score, Precision, and Recall of *PreMulBVD* are 3.34%, 4.40%, 4.60%, and 4.16%, respectively. Compared to Instruction2vec, the improvements are 8.42%, 9.21%, 10.22%, and 8.17%, respectively. Compared to VulPin, the improvements are 4.07%, 5.56%, 6.61%, and 4.29%, respectively. The main reason why *PreMulBVD* outperforms HAN-BSVD and Instruction2vec is that the two methods only consider the sequential structure of the assembly code, resulting in less information about the semantics and the code structure, while *PreMulBVD* takes into account a variety of features of the code at the same time. As for VulPin, compared to AST and pseudo-code sequences, representing binary code using slices and code blocks has the significant disadvantage of lacking high-level syntactic information. AST can clearly display the structure and syntactic relationships of the code, including rich contextual information such as variables, function calls, and control flow. In contrast, slices and code blocks usually represent only part of the code, lacking a complete syntactic and semantic perspective. Due to the absence of global context, slices and code blocks are less readable and interpretable, often requiring inference about their role in the overall program during analysis. This makes understanding the logic of the code more challenging. In fact, the types of vulnerabilities in practice are so wide-ranging that it is difficult to detect certain vulnerabilities based solely on a separate type of information alone. *PreMulBVD* incorporates structural features of the code, i.e., AST, while utilizing a pre-trained model to efficiently extract information from pseudo-code, which achieves accurate detection of multiple vulnerability types.

Next, we illustrate how *PreMulBVD* differs from the SOTA baseline methods, i.e. HAN-BSVD, with an example. Fig. 6 shows a program containing a vulnerability selected from CWE672, the cause of the flaw is that when `*i` is detected to be 0, the list will be cleared. After clearing the list, `i` becomes invalid, but the code still tries to use it (`cout << " "<< *i;`), which can lead to undefined behaviors or program crashes. According to the experimental results, the vulnerability in this program was only detected by *PreMulBVD*. The reason why HAN-BSVD was not successful is that in the assembly code corresponding to this program, the code `call clear_data`, where data is cleared, and the code `mov eax, [rdi]`, where the iterator is called again, are so far away that it is difficult to detect them by relying on the sequential structure only. In contrast, *PreMulBVD* can tell through the syntax tree structure of the AST that the two (`clear()` and the access to `i`) are dependent, and that they are placed immediately after each other so that it can be inferred that the iterator `i` was already invalidated at the time of the access. This example illustrates that our method of modeling code structure using ASTs generated from pseudo-code achieves a better vulnerabilities representation than other methods.

```
void bad()
{
  list<int> data;
  data.push_back(100);
  data.push_back(0);
  {
    list<int> ::iterator i;
    cout << "The list contains: ";
    for( i = data.begin(); i != data.end(); i++)
    {
      if (!*i)
      {
        data.clear();
      }
      /* POTENTIAL FLAW: Dereference the iterator,
which may be invalid if data is cleared */
      cout << " " << *i;
    }
    cout << endl;
  }
}
```

Fig. 6. A vulnerable program in the CWE672 directory.

Table 4
Performance of different components in *PreMulBVD*.

Setup	Accuracy	F1-Score	Precision	Recall
PreMulBVD$_{code}$	0.9737	0.9368	0.8840	**0.9962**
PreMulBVD$_{AST}$	0.9221	0.7972	0.7848	0.8100
PreMulBVD$_{code+AST}$	**0.9875**	**0.9731**	**0.9738**	0.9720

The results of RQ1 show that the binary vulnerability detection capability of *PreMulBVD* outperforms the other three baseline methods on the evaluation metrics Accuracy, F1-score, Precision, and Recall. The superiority is attributed to *PreMulBVD*'s idea of considering both code sequential semantics and code structure for vulnerability detection.

5.2. RQ2: Contribution of different components to vulnerabilities detection of *PreMulBVD*

Both pseudo-code and AST extracted from binary code are used in *PreMulBVD*, and in order to investigate the extent to which each type of information contributes to the vulnerabilities detection process, we will conduct experiments using each of the two types of information separately in RQ2. The final results of RQ2 are shown in Table 4.

PreMulBVD$_{code+AST}$ is higher than *PreMulBVD*$_{code}$ and *PreMulBVD*$_{AST}$ in three of four metrics. Specifically, compared to *PreMulBVD*$_{code}$, the improvements in Accuracy, F1-score, Precision, and Recall are 1.42%, 3.87%, 10.16%, and −2.43%, respectively. Compared to *PreMulBVD*$_{AST}$, the improvements are 7.09%, 22.06%, 24.08%, and 20.00%, respectively. Since *PreMulBVD*$_{code+AST}$ outperforms *PreMulBVD*$_{AST}$ and *PreMulBVD*$_{code}$ in almost all four metrics, it is obvious that neither pseudo-code nor AST is as effective as the combination of the two when used alone for vulnerability detection. This result shows that ASTs and pseudo-code are both needed for detecting vulnerabilities.

Using only pseudo-code for vulnerability detection, *PreMulBVD*$_{code}$ demonstrates ideal performance across four metrics, even comparable to baseline methods. This is because pseudo-code provides program control flow information, including conditional statements, loop structures, and jump instructions, which helps in understanding the program's execution path. These details enable *PreMulBVD*$_{code}$ to detect vulnerabilities in most categories. However, we can easily notice that the recall of *PreMulBVD*$_{code}$ is higher than *PreMulBVD*$_{code+AST}$, but the other three metrics are lower than *PreMulBVD*$_{code+AST}$. This is because the ratio of positive to negative samples in the dataset we used is 1:4. Due to the small number of positive samples, the model tends to classify negative samples as positive samples to improve recall, resulting in more false positive samples. On the other hand, *PreMulBVD*$_{code+AST}$ achieves a notable improvement in precision while only a minor decline in recall, further indicating that using both AST and pseudo-code

```
1  #include <climits>
2  void bad()
3  {
     /* INCIDENTAL: CWE 338 - Use of Cryptographically
4  Weak PRNG */
5    int intRand = rand();
6
7    /* FLAW: This expression is always false */
8    if (intRand < INT_MIN)
9    {
10     printLine("Never prints");
11   }
12 }
```

Fig. 7. A vulnerable program in the CWE570 directory.

Table 5
Performance of each pre-trained model.

Models	Accuracy	F1-Score	Precision	Recall
GraphCodeBERT	**0.9875**	**0.9731**	**0.9738**	**0.9720**
CodeBERT	0.9715	0.9244	0.8949	0.9557
ContraBERT	0.9719	0.9401	0.9620	0.9190
UniXcoder	0.9724	0.9331	0.9506	0.9162

together is more stable in the case of imbalanced data compared to using only pseudo-code.

At the same time, *PreMulBVD*$_{AST}$, which only uses ASTs for vulnerability detection, is worse than *PreMulBVD*$_{code}$. This is primarily because ASTs lack some information compared to pseudo-code. When using an AST to represent code, character-level details, and contextual information are lost, which can result in certain types of vulnerabilities, such as suspicious comments (CWE-546) and hardcoded passwords (CWE-259), not being represented by an AST. However, there are also many vulnerabilities that cannot be detected with pseudo-code but can be detected with ASTs, such as the example shown in Fig. 7.

In this program, the `if` module on line 7 will never be executed because the loop condition was incorrectly evaluated to be false in all cases. AST can easily represent a branch of code that will not be correctly executed, while pseudo-code is much more difficult to determine whether a branch is executed only based on the branch condition. As this example demonstrates, AST has unique advantages over pseudo-code in representing the syntactic structure, type information, variable scope, and static structure of the program.

The final experimental results of RQ2 demonstrate that both AST and pseudo-code are effective for vulnerability detection in *PreMulBVD*.

5.3. RQ3: Comparison of different pre-trained models

In order to investigate the impact of different pre-trained language models on the vulnerability detection task, we compare four different pre-trained models, as described in Section 4.4. In addition to GraphCodeBERT, which is used by default in our experiments, we also chose the classic CodeBERT and ContraBERT. The former was chosen to directly demonstrate that GraphCodeBERT can better represent ASTs compared to other BERT-like models, while the latter was chosen for an intuitive comparison between the latest BERT-like code language model with GraphCodeBERT. Additionally, we have incorporated UniXcoder to validate models of types other than CodeBERT-like models.

For a fair comparison, we follow the configuration in Section 4 to train UniXcoder, CodeBERT and ContraBERT. We set the training parameters of RQ3 uniformly based on our experience with experiments conducted on GraphCodeBERT. The batch size is 64, the learning rate is 0.00001, and the epoch is 30. The specific experimental results are shown in Table 5.

According to the experimental results, the features extracted by GraphCodeBERT are the most effective and are higher than CodeBERT and ContraBERT in terms of Accuracy, F1-score, Precision, and Recall. Specifically, compared to CodeBERT, the improvements of Accuracy, F1-score, Precision, and Recall are 1.65%, 5.27%, 8.82%, and 1.71%, respectively. Compared to ContraBERT, the improvements are 1.61%, 3.51%, 1.23%, and 5.77%, respectively. Compared to UniXcoder, the improvements are 1.55%, 4.29%, 2.44%, and 6.09%, respectively.

Compared to CodeBERT, which learns based on the simple textual information of the code (e.g., code markup and natural language comments) and only considers the sequential and semantic information of the code, GraphCodeBERT supplements the dependency information of the code by introducing a Data Flow Graph and uses Graph Neural Network (GNN) to process the code graph to model the complex dependencies (e.g., function calls, variable assignments, loops, etc.) in the code. In *PreMulBVD*, it is necessary to use AST for vulnerability detection. CodeBERT is good at representing sequential information, and it is difficult to fully understand the structural information in the AST, so its vulnerability detection capability is weaker than GraphCodeBERT, which is specially optimized for graph structures.

ContraBERT is not inferior to CodeBERT in modeling sequential code, and its innovation is to improve semantic robustness through comparative learning, which improves the stability of the model in the event of minor code changes. As a result, ContraBERT is better in F1 score and Precision metrics than CodeBERT. On the other hand, ContraBERT's results are closer to GraphCodeBERT's. Both GraphCodeBERT and ContraBERT demonstrate a significant outperformance compared to CodeBERT, optimizing semantic representation from the perspectives of code syntax structures and code sequences, respectively. However, GraphCodeBERT finally exhibits a stronger capability. The primary reason for GraphCodeBERT's superiority lies in its integration of graph structure information, where it combines the AST with data flow. The integration of graph structure allows the model to capture code semantics and structure and understand code execution paths (including the exceptional execution paths). In this way, GraphCodeBERT has an advantage in being able to model complex code relationships and detect vulnerabilities that are difficult to capture by analyzing code sequences alone, such as use-after-free (CWE-416) and deserialization of untrusted data (CWE-502). While ContraBERT achieves a notable improvement by enhancing the semantic representation of code sequences through contrastive learning, its focus remains on sequential data, making it difficult to handle vulnerabilities like CWE-416 and 502. Therefore, although their overall performance results appear similar, GraphCodeBERT's deeper structural insight makes it more effective and powerful for vulnerability detection.

UnixCoder is primarily trained on system-level commands and low-level programming, making it less effective in handling high-level programming languages. Its model design focuses on system-level tasks and lacks the capability to deeply understand the complex syntax and semantics of high-level code. Additionally, UnixCoder does not inherently support graph-based representations, which limits its ability to handle code structures that are better represented as graphs.

In conclusion, the experimental results of RQ3 show the superiority of the pre-trained model used by *PreMulBVD* for pseudo-code and ASTs representation. GraphCodeBERT is strong in both aspects. And the experimental results are in line with our expectations.

5.4. RQ4: Comparison of different decompilers.

PreMulBVD defaults to using IDA Pro as the decompiler to generate pseudo-code. In RQ4, we will analyze the impact of different decompilers on the vulnerability detection capability of *PreMulBVD*. As mentioned in Section 4.4, we will conduct experiments using IDA Pro and Ghidra as decompilers. The results of RQ4 are shown in Table 6.

In Table 6, *PreMulBVD*$_{IDA}$ represents the use of IDA Pro as the decompiler, and *PreMulBVD*$_{Ghidra}$ represents the use of Ghidra as the decompiler. The experimental results indicate that IDA Pro is slightly stronger than Ghidra in three metrics, but the overall performance difference between the two is not significant. Specifically, the differences between IDA Pro and Ghidra in metrics Accuracy, F1-score, Precision,

Table 6
Performance on different decompiler.

Setup	Accuracy	F1-Score	Precision	Recall
PreMulBVD$_{IDA}$	**0.9875**	**0.9731**	0.9738	**0.9720**
PreMulBVD$_{Ghidra}$	0.9646	0.9700	**0.9798**	0.9604

and Recall are 2.37%, 3.20%, 0.62%, and 1.21%, respectively, with the largest difference being only 3.20%.

The pseudo-code generated by IDA Pro performs better in terms of readability, variable recovery, type inference, and control flow optimization. It can more accurately infer data types, eliminate redundant code, and generate structured C code that is closer to the source code, making the analysis more intuitive. Ghidra, on the other hand, does not perform as well as IDA Pro in the recovery quality of complex functions, especially when inferring pointers, struct members, and calling conventions. It may generate more additional type casts or unoptimized code. Overall, the quality difference between the pseudo-code generated by the two decompilers is small, and the impact on the vulnerability detection of *PreMulBVD* is also minimal. This experimental result indicates that different decompilers have limited influence on *PreMulBVD*, further demonstrating the good robustness of *PreMulBVD*.

5.5. RQ5: Comparison of improvements in vulnerability detection between pseudo-code and assembly code.

PreMulBVD uses pseudo-code instead of assembly code to represent binary code. As mentioned in Section 1, pseudo-code, compared to binary code, lacks the abstraction of high-level languages and cannot represent complex logic and data structures. Therefore, its semantics are limited and it is difficult to express advanced program intentions. The aim of this RQ is to demonstrate that pseudo-code is superior to assembly code on different decompilers in binary vulnerability detection tasks. The final results of RQ5 are shown in Table 7.

Based on Table 7, it is evident that *PreMulBVD* shows significant improvements over *PreMulBVD*$_{assembly+IDA}$ and *PreMulBVD*$_{assembly+Ghidra}$ across all four metrics. Specifically, compared to *PreMulBVD*$_{assembly+IDA}$, the improvements in terms of Accuracy, F1-score, Precision, and Recall of *PreMulBVD* are 5.85%, 6.85%, 3.68%, and 9.97%, respectively. Compared to *PreMulBVD*$_{assembly+Ghidra}$, the improvements are 10.81%, 9.44%, 7.80%, and 11.09%, respectively. The result confirms that using pseudo-code to represent binary code is more advantageous than using assembly code for vulnerability detection tasks. The reason why pseudo-code offers better vulnerability detection capabilities compared to assembly code lies in two aspects. First, assembly code lacks the abstract concepts of high-level languages to represent complex logic and data structures, which limits its semantics and makes it difficult to express the intentions of high-level programs. Second, in *PreMulBVD*, pre-trained models trained using datasets of high-level language code are employed. These pre-trained models are less capable of representing assembly code compared to pseudo-code, which more closely resembles high-level language code.

Moreover, the main reason why *PreMulBVD*$_{assembly+IDA}$ performs better than *PreMulBVD*$_{assembly+Ghidra}$ is that the assembly code generated by IDA Pro is generally clearer and more optimally structured compared to that generated by Ghidra. It tends to have less instruction redundancy, higher consistency in register naming, and greater precision in identifying function boundaries, which allows it to better parse local variables and calling conventions. Additionally, IDA Pro is more intelligent in handling instruction alignment and pseudo-instructions, reducing unnecessary 'NOP' or 'JUMP' pseudo-instructions.

As we mentioned in Section 1 and analyzed in Section 2.3, assembly code has a poorer ability to represent high-level semantics, such as complex logic and data structures, compared to pseudo-code. The experimental results of RQ5 effectively demonstrate that pseudo-code significantly enhances *PreMulBVD*$_{assembly}$'s vulnerability detection capabilities compared to assembly code.

Table 7
Performance using assembly code generated from different decompiler.

Setup	Accuracy	F1-Score	Precision	Recall
PreMulBVD	**0.9875**	**0.9731**	**0.9738**	**0.9720**
PreMulBVD$_{assembly+IDA}$	0.9329	0.9107	0.9392	0.8839
PreMulBVD$_{assembly+Ghidra}$	0.8912	0.8892	0.9034	0.8750

Table 8
Performance using different optimization settings.

Setup	Accuracy	F1-Score	Precision	Recall
PreMulBVD$_{default}$	**0.9875**	**0.9731**	0.9738	**0.9720**
PreMulBVD$_{O3}$	0.9835	0.9718	**0.9807**	0.9631

5.6. RQ6: Comparison of different optimization settings.

When constructing the dataset, we compiled the source code from the Juliet Test Suite into binary code using the default optimization level. However, in practical compilation scenarios, various optimizers are used. Therefore, in this research question, we will include the highest optimization level O3 of the GCC compiler as a comparison. We will compare the differences in *PreMulBVD*'s vulnerability detection capabilities between binary code generated at the default level and at O3. The final results of RQ6 are shown in Table 8.

In Table 8, *PreMulBVD*$_{default}$ represents training and detection using binary code generated with the default optimization level, while *PreMulBVD*$_{O3}$ represents training and detection using binary code generated with the O3 optimization level. By observing the results in Table 8, it is evident that regardless of whether the binary code is generated using the default optimization level or the O3 level, *PreMulBVD* demonstrates excellent and consistent vulnerability detection capabilities. Specifically, the differences between *PreMulBVD*$_{default}$ and *PreMulBVD*$_{O3}$ in metrics Accuracy, F1-score, Precision, and Recall are 0.41%, 0.13%, 0.71%, and 0.92%, respectively, with the largest difference being only 0.92%. Therefore, based on these experimental results, we can conclude that *PreMulBVD* can effectively detect vulnerabilities in binary code generated at any optimization level.

The differences in code generated at different optimization levels mainly lie in execution efficiency and size. However, the optimization process preserves the core logic and functionality of the program. The control flow and data structures remain largely the same, with adjustments in details such as loop unrolling and function inlining. These adjustments have minimal impact on the vulnerability statements, so they do not significantly affect *PreMulBVD*'s vulnerability detection performance.

6. Threats to validity

Although *PreMulBVD* has the overall effectiveness, there are several threats:

Constructed threat. *PreMulBVD* is model-agnostic, in which the pretraining module can be replaced using a variety of pre-trained models, and in our experiments, we evaluate its portability by integrating four typical models, namely GraphCodeBERT (Guo et al., 2020), CodeBERT (Feng et al., 2020), ContraBERT (Liu et al., 2023) and UniXcoder (Guo et al., 2022). As to whether the proposed approach is applicable to other pre-trained source code models, such as CodeT5 (Wang et al., 2021), we did not explore them because they did not sufficiently match our requirements for pre-trained models.

Internal threat. *PreMulBVD* uses IDA Pro, which has been widely used, as a tool to decompile binary code. Although its effectiveness has been widely verified, it may not be able to decompile binary code completely and correctly in some cases, which may have an impact on the accuracy of vulnerability detection.

External threat. Although we have already assessed the effectiveness of *PreMulBVD* in the C and C++ programming languages, many other languages remain unevaluated due to the limitations of available datasets. To fully determine the generalizability of *PreMulBVD*, it will be essential to conduct evaluations across a broader range of programming languages in the future.

Our experiments are all based on binaries generated by the GCC compiler. The applicability of our findings derived from the evaluation may vary for binaries produced by other compilers, such as Clang or MSVC, as different compilers and optimization settings can introduce variations in the resulting binary code.

7. Related work

Current techniques for detecting vulnerabilities in static binary code can be broadly categorized into methods based on code similarity and those based on vulnerability patterns (Wang et al., 2023a).

7.1. Methods based on code similarity

Code similarity-based methods involve creating a pre-existing database of known vulnerable code snippets and then measuring the similarity between the target code and these known snippets. If the similarity surpasses a certain threshold, the target code is considered to have the same vulnerability. The most intuitive way to compute the similarity of binary functions is to use the contents of the assembly code to compute edit distances to detect similarity between functions. Khoo et al. concatenated consecutive mnemonics from assembly language into N-grams for similarity calculation (Khoo et al., 2013). After the advent of methods for directly analyzing assembly code, methods focusing on comparing the similarity of graphs began to appear (Eschweiler et al., 2016; Feng et al., 2016; Pewny et al., 2015). David et al. proposed Trecelet, which concatenates instructions from adjacent basic blocks in Control Flow Graphs (CFGs) for similarity calculation (David and Yahav, 2014), but the computational efficiency of this method is very poor. To improve the computation efficiency and increase the accuracy of the results, Feng et al. proposed Genius, which utilizes machine learning techniques for function encoding (Feng et al., 2016). Genius builds on CFGs by transforming CFGs into attributed control flow graphs (ACFGs) through feature engineering and calculates the similarity via graph embeddings generated by comparing with the codebook. Then Xu et al. proposed Gemini (Xu et al., 2017), which used a graph embedding network that evolved from Structure2vec (Dai et al., 2016) to embed ACFG into vector space, where the network parameters are trained with Siamese network architecture (Bromley et al., 1993). FIT first learns semantic features from binary code using a neural network model, which is used to screen out the potential vulnerability functions (Liang et al., 2020). SAFE extracts semantic information from assembly instructions and generates embedding representations of functions using a self-attentive neural network (Massarelli et al., 2021). VulHawk lifts binary code to an architecture-agnostic intermediate representation, uses entropy-based adapters to align function embeddings across environments, and applies a progressive search strategy (Luo et al., 2023). Asteria-Pro adopted a TreeLSTM network to summarize function semantics as function commonality and introduces domain knowledge before and after deep learning model-based function encoding to eliminate a large proportion of non-homologous functions and score homologous functions higher, separately (Yang et al., 2023).

However, these methods tend to be less generalizable, especially for cross-architecture detection, and they have difficulty extracting features that are shared across architectures (Yang et al., 2023). So for more accurate detection, semantic-based methods are proposed and applied to code similarity detection, which focus on code functionality. Feng et al. proposed extracting conditional formulas from raw binary code, lifting it to a platform-independent intermediate representation (IR), and using data-flow analysis for binary code similarity detection (Feng et al., 2017). αdiff proposed by Liu et al. uses the raw bytes of a function, the function call, and the function's import function to extract three types of semantic features for similarity detection (Liu et al., 2018).

7.2. Methods based on vulnerability patterns

However, code similarity-based methods can only detect cloned vulnerabilities, making it difficult to detect unknown vulnerabilities, and it is necessary to construct a complete vulnerability database, otherwise the accuracy of detection will be affected.

In contrast, the pattern-based methods solve both of these problems, which focus on identifying specific patterns or signatures that indicate the presence of vulnerabilities in the target code (Wang et al., 2023a). Firmalice employs symbolic execution and program-slicing techniques to extract features and proposes a novel model to detect hidden authentication bypass backdoors in target embedded devices (Shoshitaishvili et al., 2015). CWE_checker utilizes pattern matching to identify potential unknown vulnerabilities by leveraging vulnerability patterns based on Common Weakness Enumeration (CWE), which are defined by human experts (Nils-Edvin, 2019). However, the vulnerability detection capability of these methods relies on predefined patterns, which are often too general and must be extracted manually, resulting in poor detection capabilities and cumbersome processes.

Therefore, some researchers use deep learning to learn vulnerability patterns automatically. Le et al. developed MDSeqVAE, a VAE-based detection method. It embeds instructions by splitting them into opcode and instruction information, then splicing them together. VAE maximizes divergence between vulnerable and non-vulnerable code, highlighting crucial regions (Le et al., 2019). Lee et al. proposed to learn hidden patterns in assembly code and presented an instruction embedding method called Instruction2vec (Lee et al., 2019). They noted the fixed syntax of assembly code and used vector concatenation to make Instruction2vec more informative than word2vec. The learned vector representations are then used to train Text-CNN to extract features of vulnerable code. HAN-BSVD trains an instruction embedding model using BiGRU and a word-attention module. It then integrates the attention mechanism with a text-CNN to build a feature extraction network that captures local features and highlights key areas, thereby extracting more effective vulnerability features (Yan et al., 2021). Recently, some researchers have further improved the representation of features and used library/API function calls as an important determination basis for vulnerability detection. BVDetector extracts program slices related to library/API function calls at the assembly code level (Tian et al., 2020). VIVA effectively identifies both known and unknown vulnerabilities by utilizing binary program slicing techniques in combination with decompilation, which helps generate patterns for detecting vulnerabilities and patches (Xiao et al., 2021).

All of these methods focus on better representing the binary code after disassembly, in this paper, we use pre-trained models to represent the pseudo-code and ASTs disassembled from the binary code, which improves the accuracy of the representation as well as the generalization of the methods compared to the traditional methods.

8. Discussion

Decompilers may affect the performance of our proposed model. Since our method relies on pseudo-code to extract features, it can only handle binary code that can be successfully decompiled into pseudo-code and cannot be used for instruction sets that are not supported by decompilers. At the same time, current decompilers do not ensure the generation of completely correct pseudo-code, so the performance of the model is affected by the decompiler. A better decompiler can effectively improve the performance of the model.

There may be a gap between the pre-trained model and the pseudo-code. Based on our observation of the pseudo-code generated by decompilation, it contains some hardware-level functions that are not often used in high-level languages, while the pre-trained models are often trained on high-level language program datasets, which may not be able to represent these code correctly.

9. Conclusion

In this paper, we introduce *PreMulBVD*, a new approach to binary code vulnerability detection. *PreMulBVD* utilizes abstract syntax trees (ASTs) and pseudo-code to obtain the structural and semantic information of the code and further generates embeddings for them using pre-trained models. *PreMulBVD* uses the popular decompiler IDA Pro to decompile binary code. The resulting assembly code is further parsed into pseudo-code and ASTs, thus preserving both the sequential semantic and structural information of the binary code. The pseudo-code is normalized to ensure that information not related to vulnerability detection is filtered, while the ASTs are represented in the form of graphs to meet the input requirements of the deep learning model. *PreMulBVD* uses a pre-trained model, in particular, GraphCodeBERT, to obtain embeddings serving as the representation of the pseudo-code and ASTs. *PreMulBVD* combines GraphCodeBERT with Multi-Layer Perceptron (MLP) to build a classifier, which is then fine-tuned using our dataset. The fine-tuned model is used for binary vulnerability detection. Taking a function-level binary program as input, the model can determine whether the program contains a vulnerability or not.

To evaluate the effectiveness of *PreMulBVD*, we designed and conducted experiments on a commonly used dataset, the Juliet Test Suite, compared its performance with state-of-the-art methods, and investigated the influence of *PreMulBVD*'s individual components on the vulnerability detection result. Our experimental results show that *PreMulBVD* achieves high accuracy, precision, recall, and f1-score and it outperforms current state-of-the-art methods. These results demonstrate the effectiveness of *PreMulBVD* for binary vulnerability detection.

CRediT authorship contribution statement

Chenliang Xing: Writing – original draft, Validation. **Xiaoyuan Xie:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization. **Qi Xin:** Writing – review & editing, Writing – original draft. **Gong Chen:** Writing – review & editing, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (No. 62472326 and No. 62202344), and CCF-Zhipu Large Model Innovation Fund, China (No. CCF-Zhipu202408).

Data availability

Data will be made available on request.

References

Bo, L., Li, Y., Sun, X., Wu, X., Li, B., 2023. VulLoc: vulnerability localization based on inducing commits and fixing commits. Front. Comput. Sci. 17 (3), 173207.

Boudjema, E.H., Verlan, S., Mokdad, L., Faure, C., 2020. VYPER: Vulnerability detection in binary code. Secur. Priv. 3 (2), e100.

Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R., 1993. Signature verification using a" siamese" time delay neural network. Adv. Neural Inf. Process. Syst. 6.

Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C., 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proceedings of the 44th International Conference on Software Engineering. pp. 1456–1468.

Chukkol, A.H.A., Luo, S., Yunusa, H., Yusuf, A.A., Mohammed, A., 2023. VulPin: Finer-grained slicing for pinpointing vulnerability in binary programs. In: Proceedings of the 2023 7th International Conference on Computer Science and Artificial Intelligence. pp. 116–122.

CWE, 2024. CWE - common weakness enumeration. URL: https://cwe.mitre.org/.

Dai, H., Dai, B., Song, L., 2016. Discriminative embeddings of latent variable models for structured data. In: International Conference on Machine Learning. PMLR, pp. 2702–2711.

David, Y., Yahav, E., 2014. Tracelet-based code search in executables. Acm Sigplan Not. 49 (6), 349–360.

Devlin, J., 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Eschweiler, S., Yakdan, K., Gerhards-Padilla, E., et al., 2016. Discovre: Efficient cross-architecture identification of bugs in binary code. In: Ndss. vol. 52, pp. 58–79.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Feng, Q., Wang, M., Zhang, M., Zhou, R., Henderson, A., Yin, H., 2017. Extracting conditional formulas for cross-platform bug search. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security. pp. 346–359.

Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H., 2016. Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 480–491.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Hin, D., Kan, A., Chen, H., Babar, M.A., 2022. Linevd: statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 596–607.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

Khoo, W.M., Mycroft, A., Anderson, R., 2013. Rendezvous: A search engine for binary code. In: 2013 10th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 329–338.

Le, T., Nguyen, T.V., Le, T., Phung, D., Montague, P., De Vel, O., Qu, L., 2019. Maximal divergence sequential auto-encoder for binary software vulnerability detection. In: International Conference on Learning Representations 2019. International Conference on Learning Representations (ICLR).

Lee, Y., Kwon, H., Choi, S.-H., Lim, S.-H., Baek, S.H., Park, K.-W., 2019. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN. Appl. Sci. 9 (19), 4086.

Li, L., Ding, S.H., Tian, Y., Fung, B.C., Charland, P., Ou, W., Song, L., Chen, C., 2023. VulANalyzeR: Explainable binary vulnerability detection with multi-task learning and attentional graph convolution. ACM Trans. Priv. Secur. 26 (3), 1–25.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Liang, H., Xie, Z., Chen, Y., Ning, H., Wang, J., 2020. FIT: Inspect vulnerabilities in cross-architecture firmware by deep learning and bipartite matching. Comput. Secur. 99, 102032.

Lin, G., Zhang, J., Luo, W., Pan, L., De Vel, O., Montague, P., Xiang, Y., 2019. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans. Dependable Secur. Comput. 18 (5), 2469–2485.

Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W., 2018. αDiff: cross-version binary code similarity detection with dnn. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 667–678.

Liu, R., Wang, Y., Xu, H., Liu, B., Sun, J., Guo, Z., Ma, W., 2024. Source code vulnerability detection: Combining code language models and code property graphs. arXiv preprint arXiv:2404.14719.

Liu, S., Wu, B., Xie, X., Meng, G., Liu, Y., 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2476–2487.

Loshchilov, I., Hutter, F., et al., 2017. Fixing weight decay regularization in adam. 5, arXiv preprint arXiv:1711.05101.

Luo, Z., Wang, P., Wang, B., Tang, Y., Xie, W., Zhou, X., Liu, D., Lu, K., 2023. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search. In: NDSS.

Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R., 2021. Function representations for binary similarity. IEEE Trans. Dependable Secur. Comput. 19 (4), 2259–2273.

Mikolov, T., 2013. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Nie, X., Li, N., Wang, K., Wang, S., Luo, X., Wang, H., 2023. Understanding and tackling label errors in deep learning-based vulnerability detection (experience paper). In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 52–63.

Nils-Edvin, E., 2019. Hunting binary code vulnerabilities across CPU architectures.. Black Hat USA..

N.S.A., N., 2021. Ghidra. URL: https://www.nsa.gov/resources/everyone/ghidra/.

NVD, 2024. NVD - search and statistic. URL: https://nvd.nist.gov/vuln/search.

OWASP, 2024. OWASP top ten. URL: https://owasp.org/www-project-top-ten/.

Padmanabhuni, B.M., Tan, H.B.K., 2015. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In: 2015 IEEE 39th Annual Computer Software and Applications Conference. vol. 2, IEEE, pp. 450–459.

Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T., 2015. Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp. 709–724.

Radford, A., 2018. Improving language understanding by generative pre-training.

hex rays, 2019. IDA pro. URL: https://www.hex-rays.com/products/ida/index.shtml.

Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G., 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS. vol. 1, 1–1.

Tian, J., Xing, W., Li, Z., 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. Inf. Softw. Technol. 123, 106289.

Wang, Y., Jia, P., Peng, X., Huang, C., Liu, J., 2023a. BinVulDet: Detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention. Comput. Secur. 125, 103023.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859.

Wang, R., Xu, S., Tian, Y., Ji, X., Sun, X., Jiang, S., 2024. SCL-CVD: Supervised contrastive learning for code vulnerability detection via GraphCodeBERT. Comput. Secur. 145, 103994.

Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2020. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958.

Wang, Y., Ye, Y., Wu, Y., Zhang, W., Xue, Y., Liu, Y., 2023b. Comparison and evaluation of clone detection techniques with different code representations. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 332–344.

Wu, B., Zou, F., 2022. Code vulnerability detection based on deep sequence and graph models: A survey. Secur. Commun. Netw. 2022 (1), 1176898.

Xiao, Y., Xu, Z., Zhang, W., Yu, C., Liu, L., Zou, W., Yuan, Z., Liu, Y., Piao, A., Huo, W., 2021. Viva: Binary level vulnerability identification via partial signature. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 213–224.

Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D., 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 363–376.

Yan, H., Luo, S., Pan, L., Zhang, Y., 2021. HAN-BSVD: a hierarchical attention network for binary software vulnerability detection. Comput. Secur. 108, 102286.

Yang, S., Cheng, L., Zeng, Y., Lang, Z., Zhu, H., Shi, Z., 2021. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, IEEE, pp. 224–236.

Yang, S., Dong, C., Xiao, Y., Cheng, Y., Shi, Z., Li, Z., Sun, L., 2023. Asteria-pro: Enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. ACM Trans. Softw. Eng. Methodol. 33 (1), 1–40.

Zhang, Z., Li, Y., Xue, J., Mao, X., 2024. Improving fault localization with pre-training. Front. Comput. Sci. 18 (1), 181205.

Zhang, Z., Sabuncu, M., 2018. Generalized cross entropy loss for training deep neural networks with noisy labels. Adv. Neural Inf. Process. Syst. 31.