Prism: Decomposing Program Semantics for Code Clone Detection through Compilation

Haoran Li
College of Computer Science, Nankai University, Tianjin, China

Siqian Wang
College of Computer Science, Nankai University, Tianjin, China

Weihong Quan
College of Computer Science, Nankai University, Tianjin, China

Xiaoli Gong*
College of Computer Science, Nankai University, Tianjin, China

Huayou Su*
National University of Defense Technology, Changsha, China

Jin Zhang
College of Computer Science, Nankai University, Tianjin, China

ABSTRACT

Code clone detection (CCD) is of critical importance in software engineering, while semantic similarity is a key evaluation factor for CCD. The embedding technique, which represents an object using a numerical vector, is utilized to generate code representations, where code snippets with similar semantics (clone pairs) should have similar vectors. However, due to the diversity and flexibility of high-level program languages, the code representation of clone pairs may be inconsistent. Assembly code provides the program execution trace and can normalize the diversity of high-level languages in terms of the program behavior semantics. After revisiting the assembly language, we find that different assembly codes can align with the computational logic and memory access patterns of cloned pairs. Therefore, the use of multiple assembly languages can capture the behavior semantics to enhance the understanding of programs. Thus, we propose *Prism*, a new method for code clone detection fusing behavior semantics from multiple architecture assembly code, which directly captures multilingual domains' syntax and semantic information. Additionally, we introduce a multi-feature fusion strategy that leverages global information interaction to expand the representation space. This fusion process allows us to capture the complementary information from each feature and leverage the relationships between them to create a more expressive representation of the code. After testing the OJClone dataset, the *Prism* model exhibited exceptional performance with precision and recall scores of 0.999 and 0.999, respectively.

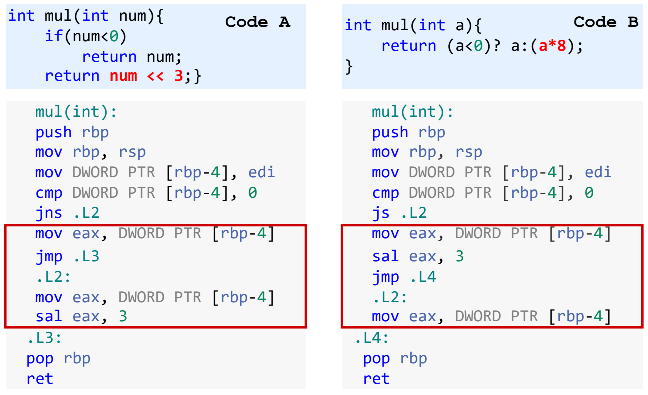
CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; • **Computer systems organization** → *Reduced instruction set computing*; *Complex instruction set computing*.

KEYWORDS

Code Clone Detection, Behavior Semantics, CISC and RISC, Feature Fusion

ACM Reference Format:
Haoran Li, Siqian Wang, Weihong Quan, Xiaoli Gong, Huayou Su, and Jin Zhang. 2024. Prism: Decomposing Program Semantics for Code Clone Detection through Compilation. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639129



```
int mul(int num){                Code A
    if(num<0)
        return num;
    return num << 3;}
```

```
mul(int):
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], edi
cmp DWORD PTR [rbp-4], 0
jns .L2
mov eax, DWORD PTR [rbp-4]
jmp .L3
.L2:
mov eax, DWORD PTR [rbp-4]
sal eax, 3
.L3:
pop rbp
ret
```

(a) Assembly for code A (x86-64)

```
int mul(int a){                  Code B
    return (a<0)? a:(a*8);
}
```

```
mul(int):
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], edi
cmp DWORD PTR [rbp-4], 0
js .L2
mov eax, DWORD PTR [rbp-4]
sal eax, 3
jmp .L4
.L2:
mov eax, DWORD PTR [rbp-4]
.L4:
pop rbp
ret
```

(b) Assembly for code B (x86-64)

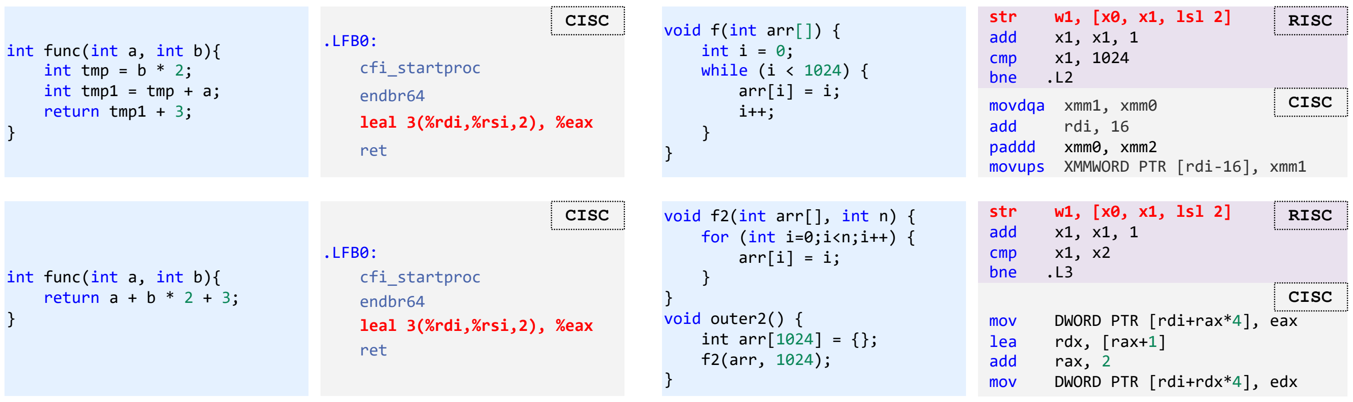
Figure 1: Motivating example. The necessity of assembly code for code clone detection.

1 INTRODUCTION

Code clone detection (CCD) is the basis of many software engineering tasks and is widely used for downstream tasks such as refactoring, code reuse, and code searching [30, 35, 42, 51, 52]. Generally, there are four types of code clones: complete cloning, parametric cloning, almost identical code snippets, and semantic cloning [7]. Semantic clones have the same or similar behavior or functionality as their counterparts but may implement using different syntax or programming constructs. As shown in the top block of Fig 1, judgment logic and multiplication use different syntax to achieve the same semantics.

Semantic clone detection is an important research topic in software engineering. It can provide a way to identify and track duplicate code in software projects, which can help protect intellectual

*Xiaoli Gong and Huayou Su are the corresponding authors. (Emails: gongxiaoli@nankai.edu.cn, shyou@nudt.edu.cn)



```
int func(int a, int b){
    int tmp = b * 2;
    int tmp1 = tmp + a;
    return tmp1 + 3;
}
```

```
CISC
.LFB0:
    cfi_startproc
    endbr64
    leal 3(%rdi,%rsi,2), %eax
    ret
```

```
int func(int a, int b){
    return a + b * 2 + 3;
}
```

```
CISC
.LFB0:
    cfi_startproc
    endbr64
    leal 3(%rdi,%rsi,2), %eax
    ret
```

(a) Calculation consistency in the CISC

```
void f(int arr[]) {
    int i = 0;
    while (i < 1024) {
        arr[i] = i;
        i++;
    }
}
```

```
RISC
str    w1, [x0, x1, lsl 2]
add    x1, x1, 1
cmp    x1, 1024
bne    .L2
CISC
movdqa  xmm1, xmm0
add     rdi, 16
paddd   xmm0, xmm2
movups  XMMWORD PTR [rdi-16], xmm1
```

```
void f2(int arr[], int n) {
    for (int i=0;i<n;i++) {
        arr[i] = i;
    }
}
void outer2() {
    int arr[1024] = {};
    f2(arr, 1024);
}
```

```
RISC
str    w1, [x0, x1, lsl 2]
add    x1, x1, 1
cmp    x1, x2
bne    .L3
CISC
mov    DWORD PTR [rdi+rax*4], eax
lea    rdx, [rax+1]
add    rax, 2
mov    DWORD PTR [rdi+rdx*4], edx
```

(b) Accessing memory consistency in RISC

Figure 2: Motivating example 2. The difference between the CISC and RISC of code snippets with the same semantics. The grey and purple blocks respectively display a subset of machine instructions generated using the CISC and RISC instruction sets.

property by identifying potential instances of code theft or copyright infringement. It can also help developers better understand and manage their codebase, making it easier to protect their intellectual property [46, 63]. Furthermore, it can facilitate the provision of elegant reference code snippets that can aid beginners in enhancing their coding skills just like Copilot [1]. In this paper, we concentrate on detecting semantic clones.

Generally, code clone detection tasks typically rely on embedding techniques [4, 13, 18, 37, 48] to generate code representations, followed by distance metric-based or deep learning-based approach for detecting whether a pair of code snippets is a clone or not. It is expected that code snippets with the same semantics possess identical embeddings. Both syntax and semantics are crucial features used to represent code. [19, 32]. To extract this information and comprehend code from high-level programming languages, a range of static analysis techniques have been utilized in prior research, such as abstract syntax tree (AST) [41, 49, 60], control flow graph (CFG) [25, 65], and data flow [8, 28].

Due to the flexibility of programming languages, it is challenging to align the code representations of clone pairs in semantic space. As shown in top part of Fig. 1 , "num << 3" in *CodeA* can be changed into "$a * 8$" in *CodeB*. Due to variations in program structures and syntax, utilizing AST and CFG to extract code features for code representation may produce inconsistent embedding. This inconsistency significantly impacts the performance of detection.

Lots of the *"semantic clone"* problems can be effectively eliminated at the assembly level with the help of compilers and various code optimization tools. As can be seen in Fig.1, when processing the assembly code generated by compilers, there is only a slight difference in the red rectangles. The assembly code provides a representation of interactions between the CPU and memory system (via load/store instructions), as well as information about program execution [6] such as arithmetic logic and control logic. Thus, assembly code can be utilized to normalize the diversity of high-level languages.

Moreover, assembly codes from different Instruction Set Architectures (ISAs) possess distinct advantages. Compiler optimizations leverage the specific characteristics of assembly languages to align the behavior of high-level languages. This provides us with additional opportunities to comprehend programs. As there are specific instructions for customized calculation operations in the complex instruction set computing (CISC) ISAs, the CISC assembly code provides more information to understand the *arithmetic logic* programs. For example, Fig. 2 (a) shows an arithmetic operation of multiplication and addition with different codes. The *leal* instruction can normalize the literal and syntactic difference of high-level language since the compiler prefers to generate fewer instructions in the optimization stages at specific optimization levels and types. Moreover, reduced instruction set computing (RISC) architectures typically support larger memory spaces, facilitating efficient memory access. As RISC instructions are relatively simple, compiler optimization techniques like data flow analysis can still be applied to eliminate differences and improve code clone detection, as shown in the RISC block of Fig. 2 (b). This simple instruction sequence can accurately capture the consistency in memory behavior with the *load* and *store* instruction, therefore suitable for understanding the memory access of programs.

It can be seen that the code execution trace in assembly code provides us with *behavior semantics* in view of program calculation and memory behavior, which are helpful in code clone detection. The compiler flexibly optimizes the code utilizing diverse and suitable instructions. However, it is not enough for a single ISA to fully represent the characteristics of program semantics. For example, the semantic clone pairs for assigning values to arrays are caused by different code styles in Fig. 2 (b). Since directly defining the size of the array, the compiler used vectorized instruction paddd to optimize the code resulting in different instructions generated. At the same time, we can also align the behavior of the program in the memory access view. Therefore, we *emphasize* that the combination of different ISAs can complement each other and enhance the understanding of semantics.

In this paper, we propose *Prism*, a novel code clone detection method by fusing behavior semantics from CISC and RISC to solve the *semantic clone* problem effectively. Specifically, we utilize the

GCC compiler to compile source code into the assembly code of CISC and RISC (Sec. 4.3). Subsequently, we employ the *Asm2vec* [14] method to generate two types of embedding for the assembly code (Sec. 3.2). We designed a code detection pretext task to fine-tune a generic language model [13] and generate task-specific code representations that are better suited for clone detection tasks (Sec. 3.3). To further refine and enhance code representation, we proposed a novel feature fusion strategy (Sec. 3.4). Given the limited expression space of assembly code, we expanded the feature space of assembly code representations. Additionally, we generated a trainable kernel using the source code embedding to interact with the assembly representation space. Our fusion strategy process enables us to capture higher-order relationships between the features that may not be apparent from local analysis of each feature in isolation. Comprehensive experiments are conducted based on the OJClone dataset [40]. The result shows that *Prism* is better than the FCDetector method [17] in terms of CCD performance. Specifically, the precision, recall, and F1 rate are improved by 2.9%, 4.9%, and 3.9%, respectively.

The main contributions of this paper are as follows:

(1) We have carefully investigated the effects of assembly code from different instruction set architectures (ISAs) on code clone detection tasks. We emphasize that assembly code from different ISAs has different representations of the memory and algebraic logic behaviors, which can be used to differentiate the semantic similarity of code clones.

(2) We present *Prism*, a code clone detection method based on a fusion of multiple features from different representations of program code, including assembly code from CISC and RISC, as well as high-level language source code. The fusion strategy leverages high-order interaction to enhance the code representation by expanding the representation space.

(3) Comprehensive experiments have been conducted on the OJClone and GCJ datasets to demonstrate the effectiveness of the proposed *Prism*. The results show that *Prism* achieves the best performance in code clone detection compared with 7 CCD solutions and 4 generic code representation models. Our code is available at https://github.com/NKU-EmbeddedSystem/Prism.

The rest of the paper is organized as follows. Sec 2 describes the background and motivation for this paper. Sec 3 illustrates our framework for *Prism*. Sec 4 presents our experimental methodology and OJClone dataset. Sec 5 demonstrates the effectiveness of the proposed *Prism* and analyzes each module of the *Prism*. Sec 6 provides the related work. Sec 7 discusses the time performance and generalization of *Prism*. Sec 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Semantic Align with Assembly Code

Code clone detection aims to identify similarities between pairs of code. In code clone detection tasks, code is transformed into embedding using code representation methods [18, 20, 29]. Similar code should correspond to similar embedding. The key challenge in generating code representations is identifying syntactic and semantic features that better describe the code [17, 25]. For instance, syntax information can be extracted using AST [41, 64], while control flow information can be captured using CFG [8, 17].

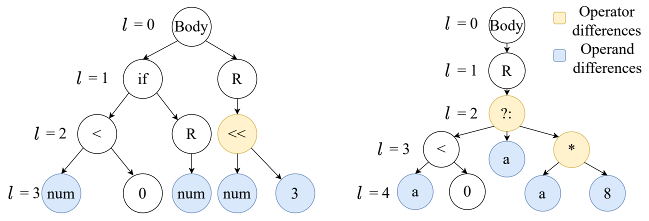


Figure 3: AST for code A and code B in Fig.1.

However, the methods based on AST and CFG may not be sufficient for identifying semantic clones. This is because the diversity of programming languages and coding styles can result in different implementations of the same functionality. As the example in Fig. 3 shows, the *CodeA* and *CodeB* in Fig. 1 share the same semantics but have completely non-comparable AST, including the levels of the AST and the parent and sibling nodes of each node. As a result, the representation of the program is easily misleading when only relying on the information extraction of the high-level language.

The instructions in assembly code are the smallest unit of program execution. Based on the information provided about the behavior of the program in computer architecture, we can normalize the diversity of high-level languages. For example, statements "$num << 3$" and "$a * 8$", though literally different, execute the same arithmetic left-shift operation after the compilers' optimization. Inspired by this, we introduced the assembly language for CCD.

2.2 Different Instruction Set Architectures

The Instruction Set Architecture (ISA) serves as the interface between hardware and software. It also forms the basic unit for defining the semantics of a program, with each instruction in assembly code serving as the equivalent of a word in a text for natural language processing tasks. Generally, ISA can be classified into two categories: CISC and RISC based on the complexity of the instruction systems [43]. Therefore, we investigated the differences between CISC and RISC architectures in terms of their ability to represent program semantics.

CISC architecture supports a rich and complex set of computational instructions, such as vector, encryption, and bit manipulation instructions. Compilers prefer to optimize code by emitting fewer instructions that leverage the specialized computing capabilities of CISC. For example, in Fig. 2(a), the compiler generates the *leal* instruction after analyzing the semantics of the expression involving multiplication and addition, regardless of the operator precedence. This enables the compiler to unify different high-level language expressions with the same computational effect into a single, similar instruction. Thus, CISC architecture can better align with algebraic logic behavior. On the contrary, CISC architectures typically offer an abundant range of memory access modes to support various types of memory, such as accessing memory with an offset based on a given address, direct access to global memory addresses, or other similar modes. Consequently, multiple types of instructions can be generated to represent the similar memory access operation in high-level source codes, which leads to a multitude of program

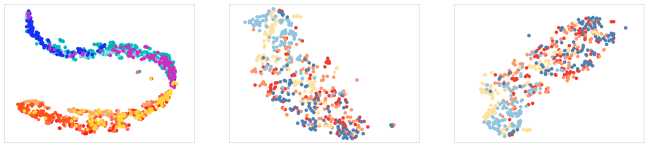

(a) CISC and RISC (b) Addition (c) concatenation

Figure 4: t-SNE visualization of code representation. (a) displays the CISC and RISC representations from 5 tasks of the OJclone in a unified code representation space. The cool-colored points represent CISC embedding, and the warm-colored points represent RISC embedding. (b) and (c) shows the visualization of code representation from the fusion of CISC and RISC embedding.

expressions that obscure the consistent memory access behavior, resulting in difficulty in aligning.

RISC architecture is designed to decouple memory and computational operations, providing explicit memory operations. This approach employs a reduced instruction set to describe program execution logic, unlike CISC, which incorporates proprietary instructions. As a result, RISC architecture provides a flexible interface corresponding to high-level language expressions. However, when describing program behavior, it leads to diverse assembly code expressions as the high-level source code changes, which obscure program behavior understanding. Nevertheless, this decoupling of memory and computational operations in RISC architecture allows for *load* and *store* operations for memory to be more explicitly defined. This results in a clearer representation of memory behavior for the program, improving its alignment with memory access behavior for clone pairs. Examples can be seen in Fig. 2(b) that consistent memory access instructions are generated even when compiling different high-level programming languages.

The aforementioned theoretical findings are also observed in our empirical study. We selected the first five tasks (Task ID: 1-5) from the OJClone dataset [40], each containing 100 semantic cloned files. We used the GCC cross-compiler to compile them into CISC and RISC assembly codes correspondingly. We then employed the *Asm2vec* model to generate a 96-dimensional embedding for the assembly codes, which are then converted into a 2-D representation using t-SNE [53]. Fig. 4 (a) shows the resulting 2-D code representations for each code file, with cool-coloured points representing CISC embedding and warm-coloured points representing RISC embedding. As can be seen from the spatial distribution of the two types of assembly representations, they exhibit linear separability. This suggests that the CISC and RISC code representations of the same high-level language code have distinct features in a single unified semantic space. It stems from their different algebraic logic and memory access patterns for the same code segment. To capture the underlying semantic behavior of the code, *Prism* requires a better fusion of this distinctive feature.

2.3 Representation in Assembly code

The diversity of the code statement is advantageous for extracting semantic features and more suitable for code representation [44]. Due to the fixed and concise code statements of assembly language, there are certain limitations for code representation. As shown in Fig. 4 (a), We find that the representations of CISC and RISC are separable, but the representations of the 5 tasks, even in CISC and RISC assembly code, are mixed together. This can lead to semantic overlap between different tasks, as there are a limited number of ISAs and statements.

We selected common feature fusion methods, addition, and concatenation, in the CCD task [17]. Specifically, we added the embedding of CISC and RISC or concatenated them directly as code representations. Next, we employed the t-SNE method to visualize the fused code representations, as shown in Fig. 4 (b) and (c). The plot includes 5 coloured points representing 5 distinct tasks. It is evident that the representations remain scattered and unclustered across tasks, indicating that previous methods have difficulty distinguishing between tasks. To enhance assembly code representation, we propose fusing features to incorporate more information. Specifically, We improve the expressiveness of assembly code by expanding and refining its representation space through high-order interaction. Further details can be found in Section 3.4.

2.4 NLP Models for Programming Languages

Introduced in 2017, Transformers [54] with a novel attention mechanism outperformed previous RNNs [12, 23] and quickly became widely adopted for natural language tasks. This breakthrough led to the development of pre-trained models such as BERT [13] and GPT [9], which have demonstrated exceptional performance in various text understanding benchmarks [47, 57], enabling NLP tasks to achieve unprecedented levels of performance. BERT is an optimized representation model based on the Transformer architecture. Due to its outstanding performance in semantic understanding, many researchers have employed it for intelligent code analysis, including CodeBERT [18], CBERT [11], and CuBERT [28]. These models learn effective representational knowledge from large amounts of code text, which can then be fine-tuned for use in various downstream software engineering tasks, such as code clone detection and method name prediction.

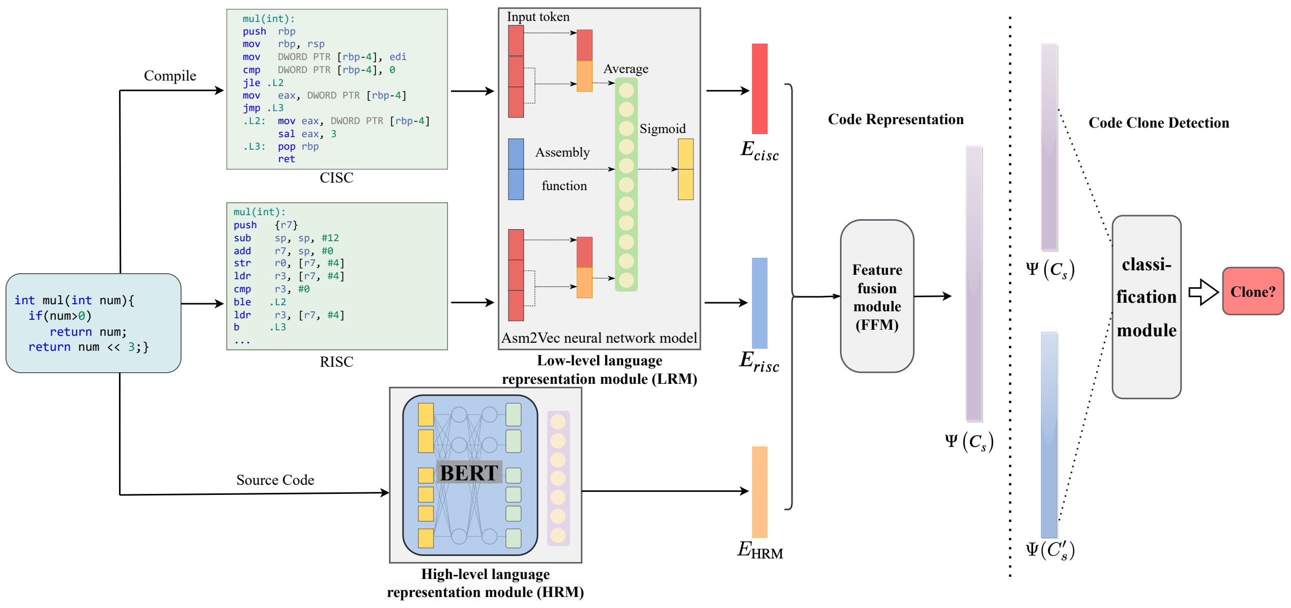
Inspired by the success of BERT in NLP, we have leveraged the model's generality and utilized *the bert-base-ucased* model to capture both syntactic and semantic knowledge of the code for generating code representations. Further details can be found in Section 3.3.

3 PRISM FRAMEWORK DESIGN

3.1 Framework Overview

The framework of *Prism* is presented in Figure 5. *Prism* extracts syntax and semantic information from low-level assembly and high-level source codes and fuses it for code clone detection. The *Prism* model comprises three modules: the low-level language representation module (*LRM*), the high-level language representation module (*HRM*), and the feature fusion module (*FFM*).

The input of *Prism* is source code written in a high-level programming language. The source code is then compiled into assembly code that supports different ISAs, such as CISC and RISC. For the *LRM*, we use an assembly feature encoder to generate behavior semantic embeddings from the assembly code. We can use any existing assembly code encoder, and we choose *Asm2Vec* [15] in our



```
int mul(int num){
  if(num>0)
     return num;
  return num << 3;}
```

```
  mul(int):
  push  rbp
  mov   rbp, rsp
  mov   DWORD PTR [rbp-4], edi
  cmp   DWORD PTR [rbp-4], 0
  jle .L2
  mov   eax, DWORD PTR [rbp-4]
  jmp .L3
  .L2:  mov eax, DWORD PTR [rbp-4]
        sal eax, 3
  .L3:  pop rbp
        ret
```

```
mul(int):
push   {r7}
sub    sp, sp, #12
add    r7, sp, #0
str    r0, [r7, #4]
ldr    r3, [r7, #4]
cmp    r3, #0
ble   .L2
ldr    r3, [r7, #4]
b     .L3
...
```

Figure 5: Overall structure of our *Prism*. The *Prism* model comprises three modules: the low-level language representation Module (*LRM*), the high-level language representation Module (*HRM*), and the Feature Fusion Module (*FFM*).

implementation. As shown in Figure 5, the semantic embeddings are denoted as E_{cisc} and E_{risc} for different ISAs. We also employ the *HRM* to generate the embedding, denoted as E_{HRM}, from the high-level source code. The low-level code embedding and high-level embedding are fused by the feature fusion module to form the code representation of the source code, which is denoted as $\Psi(C_s)$ in the Figure 5,

$$\Psi\,(C_s) = FFM[E_{\mathrm{cisc}}\,, E_{risc}, E_{\mathrm{HRM}}]. \tag{1}$$

At last, we use a classifier based on the DNN model to detect code clones. The overall clone detection model can be represented formally as follows:

$$\mathrm{F}(\Psi(C_s), \Psi(C_s')) = \begin{cases} 1 & C_s, C_s' \in \mathcal{T} \\ 0 & C_s, C_s' \notin \mathcal{T} \end{cases} \tag{2}$$

Where C_s, C_s' are two source code snips, ψ is the *Prism* code representation framework for generating the embedding based on the source code, F is a shallow classifier which is a two-layer linear model. $\mathcal{T}$ means task domain, the output of the classification module is 1 when two source codes are from the same task domain and 0 when they are from a different task domain.

3.2 Low-level Language Representation Module

We use the *Asm2Vec* method [14] in the low-level language representation module (*LRM*) of *Prism* to obtain semantic embedding from assembly code without bells and whistles. In *Asm2Vec*, assembly code is converted into a sequence by edge sampling and random walks. Then, we apply the *Asm2Vec* neural model to get the semantic embedding from the sequences. The semantic embedding is the vector to be learned during the model training procedure. In the *Asm2Vec* neural model, similar to the PV-DM model [33], we use a sliding window to scan each sequence from the beginning. At each step, the model predicts the vector of the center sequence with the neighbor sequences and semantic embedding. The sequence vector and semantic embedding are updated according to the loss function. We update the semantic embedding for assembly code until the end of the training. The vectors provided by the neighbor sequences capture behavior semantic relations, enhancing the semantic representation [15].

Specifically, We select ARM and X86 as the target assembly platforms in the *LRM* and the GCC cross-compiler as the assembly code generator and dump the assembly code with the parameter -S. Also, the function names within the corresponding assembly programs are extracted from the symbol table as training parameters for the *Asm2Vec* model. Finally, The assembly code is fed into the *LRM* to generate the $E_{\mathrm{cisc}}\,, E_{risc}$.

3.3 High-level Language Representation Module

Operands and operators are important features of the source code. Understanding semantics is also critical for CCD. In the high-level programming language representation module (*HRM*) of *Prism*, we aim to extract the syntactic and semantic features from the source code. Recent code representation models show significant advantages in understanding code [11, 18, 20]. However, we observe that they are may not well-suited for code clone detection tasks. For instance, CodeBERT[18] requires inputs in the form of comments describing the code, which are not available in CCD tasks. Moreover, GraphCodeBERT[20] relies on data flow analysis of the source code, which incurs additional time overhead. In simple terms, by utilizing a general language model (e.g., BERT [13]), we can obtain syntactic

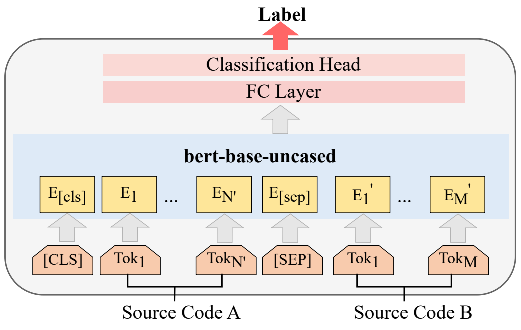


Figure 6: Fine-tuning for *HRM*. We add a classification head and an FC layer to *HRM*. After fine-tuning, the classification head is stripped off directly. *HRM* is used to generate the code representation.

features of high-level programming languages. Moreover, we devise a strategy to efficiently obtain code representations for programs consisting of syntactic and semantic information.

We make two modifications to improve the effectiveness of the generic language model: 1) *Natural language (NL) knowledge transfer to program language (PL) domain.* The generic language model has obvious advantages for natural language understanding, but the domain knowledge gap may lead to some limitations in its application to code understanding tasks. In *Prism*, we fine-tuned the generic language model using the PL dataset from the downstream task to align the NL and PL data distributions. 2) *Generate task-level code representations.* Programs often contain numerous nested control flows, function calls, and intricate data structures. However, utilizing token-level and sentence-level code representations generated by fine-tuning tasks like masked language model (MLM) or next sentence prediction (NSP) has limitations in comprehending program semantics [13, 58]. Additionally, fine-tuning models with such MLM or NSP tasks incurs a substantial computational cost, exacerbating the difficulty of adapting to smaller datasets like the PL dataset, where altering the data distribution of the trained model is more challenging. Inspired by the text classification task in NLP [13, 50], Thus, we use a text-level task to fine-tune the model, enabling the trained model to perceive the overall semantics of the program better. We also mitigate the computational burden by freezing the backbone. As a result, we obtain task-level code representations with remarkably low parameter updates.

Specifically, the *HRM* module is shown in Fig.5. We use *bert-base-uncased* [13], a pre-trained generic representation model provided by Google, as the backbone of *HRM*. Additionally, a fully connected (FC) layer is involved following the BERT model to change the output dimension. In this way, we can adjust the output dimension of BERT to collaborate better with the *LRM*. Then, we conduct the fine-tuning on *HRM* [13, 50], the framework is shown in Fig. 6. Two labelled code snips from the CCD task are fed into the *HRM* to facilitate the semantics understanding from the task level.

During fine-tuning for *HRM*, we adopt a strategy of freezing the parameters of the backbone and training only the FC layer in a purely monolingual setting specific to the target task [34, 45]. By doing so, we efficiently align the data distribution of NL and PL while significantly reducing the computational cost. Parameter statistics show that the non-frozen parameters account for a mere 0.07% of the total network model parameters. After fine-tuning, we discard the classification head and generate E_{HRM} using *HRM*.

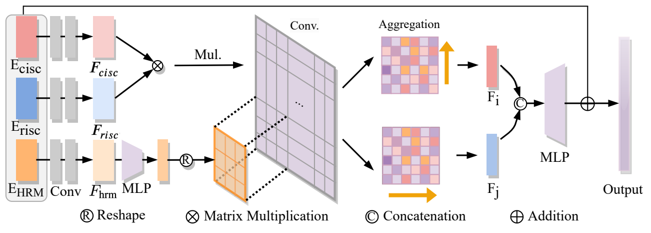


Figure 7: The overview of feature fusion module.

3.4 Feature Fusion Module

The feature fusion module to fuse the aforementioned E_{cisc}, E_{risc} and E_{HRM} into a dense vector. Specifically, We define a multi-feature hybrid vector, $E = [e_1, e_2 ..., e_n]$, where $e_n \in \mathbb{R}^{1\times l}$ denotes the embedding from specific fields, where n denotes the number of fields, l is the length of embedding. As for Prism, the number of fields is 3. We obtain the final representation $\Psi(C_s)$ by fusing vectors through a multi-stage feature interaction strategy.

A straightforward approach to obtain a fusion representation is to add those vectors directly, which can be formulated as: $E_{add} = [e_1^1 + e_2^1 + ... + e_n^1, e_1^2 + e_2^2 + ... + e_n^2, ..., e_1^l + e_2^l + ... + e_n^l]$, where e_n^l denotes the l-th element from the n-th field, l is the *index* dimension of e in field. Then we extend the length of e_1 to "$l * n$" by padding zeros: $[e_1, 0..., 0]$. The concatenate can be formulated as a special addition: $E_{concat} = [e_1|e_2|...|e_n] = [e_1|0|...|0] + [0|e_2|...|0] + [0|0|...|e_n]$. It can be observed that existing approaches for code clone detection focus solely on the interaction of elements at the same index dimension. This limitation can hinder the enlargement of code representation distance in the feature space [26, 62].

To address this issue, the *key point* of our proposed fusion module is to generate multi-stage feature interaction, thereby obtaining more discriminative code representations. In the first stage, we leverage multi-feature multiplication between the low-level code representation to expand the feature space. We then utilize a learnable kernel from the high-level code representation, which further dynamically enhances the representation of the code.

Specifically, for the first stage, the original E_{cisc}, E_{risc} and E_{HRM} are computed by convolution and marked as F_{cisc}, F_{risc} and F_{hrm}, where they are all the 1-D embedding $F \in \mathbb{R}^{1\times k}$, where k is the length of F, as shown in Fig 7. we establish a global point-wise interaction between features of F_{cisc} and F_{risc}. After the first stage, we get a matrix with the feature interaction of F_{cisc} and F_{risc}.

$$M_s(i, j) = LeakyReLU(F_{cisc}(i)^T \times F_{risc}(j)), \tag{3}$$

where $\times$ denotes the matrix multiplication, where i and j stand for the i-th and j-th location in semantic fusion matrix $M_s \in \mathbb{R}^{k\times k}$. It is noteworthy that we propose the use of the *LeakyReLU* activation function in our work [38] instead of *ReLU*, which maximizes the preservation of the original feature information.

This fine-grained interaction strategy enables interaction between feature elements to expand the representation space. Moreover, relying solely on matrix multiplication for feature fusion decreases the non-linear expressiveness of the features. To overcome this limitation, we introduce a learnable convolution kernel to perform the second stage feature fusion. Specifically, F_{hrm} is provided as input to an FC layer, which generates a reduced-dimensional vector that is subsequently reshaped into a convolutional kernel $K \in \mathbb{R}^{3\times3}$. Subsequently, we calculate the semantic fusion matrix M_s with the 3×3 convolution kernel K, resulting in a fusion feature matrix $M_f \in \mathbb{R}^{k\times k}$. This operation can be expressed as follows:

$$M_f = M_s * K = M_s * Reshape(f(F_{hrm})), \tag{4}$$

where $*$ denotes the convolution. $Reshape(\cdot)$ reformulates a 1-D feature into a 2-D convolution kernel, and $f(\cdot)$ denotes the fully-connected layer.

After two stages, the three features enable the interaction of global information. Using M_f directly as the final code representation can result in significant computational costs. Finally, we aggregated across-dimensions information for the fusion feature map M_f. we use global average pooling methods to squeeze the matrix M_f along the i and j dimensions as the following:

$$\begin{aligned} F_i = AvgPool(M_f^{i=1,2,..,k}), F_j = AvgPool(M_f^{j=1,2,..,k}) \\ \Psi(C_s) = f(concat(F_i|F_j)) + concat(E_{\text{cisc}}|E_{risc}|E_{\text{HRM}}), \end{aligned} \tag{5}$$

where $AvgPool$ denotes the 1-D average pooling layer which step is the length of M_f, $F_i \in \mathbb{R}^{1\times k}$ and $F_j \in \mathbb{R}^{1\times k}$ are the aggregate information from different dimensions. This process encapsulates information while also reducing the volume of code representation. Then F_i and F_j are concatenated and fed into the FC layer to match the vector dimensions. Residual learning [21] is also applied to generate the final code representation.

Compared to the previous index-dimensional fusion approach such as addition and concatenation, our method achieves high-order interactions through multi-stage feature interaction, thereby effectively generating discriminative code representations.

4 IMPLEMENTATION

4.1 Experiment Setup

The dimensions of E_{risc}, E_{cisc}, E_{HRM} and $\Psi(C_s)$ are 96, 96, 96, 288. There are only two hidden layers in the classification module, with the neurons of each hidden layer being 288 and 2. To address the overfitting issue, a dropout layer [22] is integrated into the classification module. For the training process, the training epoch is set to 100, the learning rate is 0.001, the batch size is 512, the optimizer is Adam [31], and the probability of dropout is 0.5. The training loss function used the cross-entropy loss function, which illustrated as Equ. 6.

$$\mathcal{L} = -\sum_{i}^{N} \left[y \log(\hat{y}) + (1-y) \log(1-\hat{y}) \right] \tag{6}$$

where N denotes the total number of samples, y denotes the true value of the ith sample, and $\hat{y}$ denotes the predicted value of the ith sample.

All the experiments are conducted on a workstation equipped with Intel Core i7-6700 CPU, 32GB memory, and an Nvidia GeForce GTX 2080Ti GPU.

4.2 Dataset Selection and Pre-Processing

In this paper, we adopt the OJClone [40] dataset to evaluate the new method we propose. This open-source dataset is a collection of C/C++ program codes submitted by different users for the same program task, conforming to the definition of type-4 semantic clones [7]. OJClone contains 52,000 files, including 104 tasks and each task contains 500 source codes.

Dataset for CCD. We removed comments, blank lines, and other irrelevant information from the original program to avoid interference with the *HRM*. To ensure a balanced distribution of labels, we generated cloned pairs within the task and non-cloned pairs across tasks for a given code program. Therefore, dataset building typically involves under-sampling. we can generate $500 \times 2 \times 52,000$ items to train the model using the entire dataset, it incurs substantial time costs. Taking the training costs and fairness comparisons into consideration, we adopt the same experimental setup as FCDetector [17]. We select 35 program tasks from the OJClone dataset and then randomly extract 100 unique programs for each task. Finally, we generated 700,000 data pairs, with the ratio of cloned pairs to non-cloned pairs being 1:1. Among them, 560,000 code pairs are used for training, and the remaining pairs for testing.

Dataset for fine-tuning High-level Language Representation Module. Similar to the format of MRPC [16], a classical text classification dataset, that of our dataset is $< C_s, C_s', label >$, where $label = 1$ denoting instances from the same task domain, and $label = 0$ denoting instances from distinct task domains. Moreover, the ratio of cloned pairs to non-cloned pairs is set to 1:1 to keep the data independently and identically distributed. Based on the overall data size of the MRPC dataset, 7000 code pairs are built as training data that do not overlap with the CCD test set. The training loss function used the cross-entropy loss function as Equ 6.

4.3 Code Compilation Details

In the process of detecting code clones using Prism, it is crucial to obtain assembly code, which includes both CISC and RISC. With the ongoing advancement of compiler technology, the same code can be compiled for multiple architectures using cross-compilers, such as cross-tool chains based on GCC [2] or cross-architecture backends implemented in LLVM. These tools enable the generation of assembly code for different architectures with different levels of optimizations. Standard compilers rely on optimization levels that represent fixed optimization sequences [39]. This provides a stable foundation for subsequent code embedding processes.

Furthermore, when processing code snippets, Prism may fail to compile due to missing variables or functions. To address this issue, the target code snippets can be packed as a function while the missing items can be complemented as external items, local variables, or function parameters. Notably, these processes can be automatically accomplished similar to the `Extract Method` provided by IDE tools such as Eclipse [3]. In this way, we can make the snippets compile while the corresponding assembly code can

Table 1: Comparison of experiment results from different methods on OJClone. The bold numbers are the best result."Syn" and "Sem" mean the method based on syntactic and semantic information. "Fusion" means fusion multi-programming language.

Method	Prec.	Recall	F1	Syn.	Sem.	Fusion
DECKARD	0.990	0.050	0.100	✓	×	×
CDLH	0.470	0.730	0.570	✓	×	×
DeepSim (ESEC'2018)	0.710	0.830	0.760	✓	✓	×
ASTNN (ICSE'2019)	0.980	0.920	0.950	✓	×	×
FCCA (TR'2020)	0.941	0.873	0.906	✓	✓	×
FA-AST (SANER'2020)	0.947	0.918	0.932	✓	✓	×
FCDetector (ISSTA'2020)	0.970	0.950	0.960	✓	✓	×
Code2vec (POPL'2019)	0.560	0.690	0.610	✓	×	×
BERT (NAACL'2019)	0.975	0.918	0.942	✓	✓	×
InferCode (ICSE'2021)	0.952	0.903	0.927	✓	×	×
GraphCodeBert (ICLR'21)	0.996	0.977	0.988	✓	✓	×
HELoC (ICPC'2022)	0.995	0.966	0.980	✓	✓	×
Prism	**0.999**	**0.999**	**0.999**	✓	✓	✓

be extracted with the external items as stubs, which has minimal impact on the semantics of the program. The stubs will be resolved at linking time, which is not relative to the semantic extraction procedure in this paper.

5 EVALUATION AND ANALYSIS

5.1 Comparison of Detection Results

Similarly to previous research [17], We have involved 7 task-specific code clone detection approaches for performance comparison, which are DECKARD [27], CDLH [59], DeepSim [65], ASTNN [64], FCCA [25], FA-AST [56] and FCDetector [17]. We use *precision*, *recall* and *F1* as three evaluation metrics.

The experiment results are shown in Tab. 1. *Prism* achieves the best performance, with the precision, recall, and F1 rate being 0.999, 0.999, and 0.999, respectively. The result shows state-of-the-art in all three evaluation metrics. Although the DECKARD [27] can achieve a similar precision due to the strict feature extraction and detection mechanism, it delivers a low recall and F1 score. FCDetector significantly improves the performance compared to the previous methods due to the fusion of syntax and semantics for joint training. However, the flexibility of high-level language can mislead the code representation, which limits the performance.

To further understand the performance of *Prism*, we have also evaluated the code representation module of *Prism* by comparing it with Code2vec [5], InferCode [10], GraphCodeBert [20] and HELoC [58] three generic code embedding approaches. We also employed the *bert-base-uncased* [13] model as another baseline. The experiment results in Tab 1 show that our *Prism* outperforms all baseline methods. Based on the pre-train representation model, our approach has a better performance than theirs. Particularly, in terms of recall, *Prism* is 3.3% more effective than HELoC. Moreover, fusing behavior semantics from multiple assembly languages, *Prism* also outperforms the *bert-base-uncased* model.

Table 2: Ablation study on effects of behavior semantics.

ID	E_{LAM}	E_{risc}	E_{cisc}	Prec.	Recall	F1
1	✓	✓	✓	**0.990**	**0.984**	**0.987**
2	✓	×	×	0.983	0.926	0.954
3	×	✓	✓	0.989	0.940	0.964
4	✓	✓	×	0.986	0.949	0.964
5	✓	×	✓	0.986	0.956	0.971

We have compared *Prism* with the other code clone detection methods, and the details are shown in Tab. 1. We mainly focus on syntactic information (such as AST) extraction, the semantics of source code(such as CFG, data flow, MLM pretext task), and the fusion code representation. (i.e. high-level program language and assembly code). To our best knowledge, the *Prism* is the first code clone detection method that uses both high-level and low-level programming languages. It can involve both syntactic and underlying semantic knowledge.

5.2 The Effectiveness of Behavior Semantics

To demonstrate the effectiveness of behavior semantics in *Prism*, we performed some ablation studies in this subsection. For fairness and simplicity, we fuse features with the concatenation method and use a classifier with 5 hidden layers, with the number of neurons in each layer being $l/2l/4l/2l/l$, where l is the length of input embedding. Tab. 2 shows the performance on clone detection with different ablation code representations in *Prism*.

First, we validate the effectiveness of behavior semantics. From the experiment result of ID 2, it can be seen that without the behavior semantics, the performance of *Prism* decreased by 5.8% in the recall. The ID 3 experiment shows that only using behavior semantics alone can yield good performance, highlighting the strong influence of behavior semantics in code clone detection tasks.

Second, we study the influence of E_{risc} or E_{cisc}. We built two ablation models in experiments ID 4 and ID 5. Comparing the two ablation models with the *Prism* model, we can observe that removing the E_{cisc} hurts the recall for about 3.5%, and removing the E_{risc} hurts the recall for about 2.8%. The experimental results show that compared with the RISC, the CISC has more diversity in the statement expression, which is advantageous for extracting semantic features. Simultaneously, the combination of multiple architecture assembly semantics optimizes the performance of code clone detection.

5.3 Comparison between IR and Assembly

Compiler-generated intermediate representation(IR) is also employed in code clone detection in recent research [55]. To demonstrate the additional performance improvement by involving assembly code, we have conducted experiments on two groups (GR), as shown in Tab 3. The LLVM-12 compiler is leveraged to generate the LLVM IR code under the `O0` optimization level. Then, we use IR2vec [55] to generate two types of code representation (marked as E_{IRf} and E_{IRs}) for IR, namely Flow-Aware (FA) and symbolic (SYM), which are provided in IR2vec. After that, different feature embeddings are concatenated and fed into the 5-layer classifier mentioned in Section 5.2.

Table 3: Comparative study between IR and assembly code for code clone detection. "ref" means reference experiments.

GR	ID	Methods	Code Repre.	Prec.	Recall	F1
I	1	IR2vec-FA	E_{IRf}	0.950	0.880	0.914
	2	IR2vec-SYM	E_{IRs}	0.954	0.878	0.914
	3	Asm2vec	E_{risc}	0.981	0.938	0.959
	4	Asm2vec	E_{cisc}	0.985	0.939	0.962
II	5	HRM+IR2vec-FA	$E_{HRM}\|E_{IRf}$	0.985	0.931	0.957
	6	HRM+IR2vec-SYM	$E_{HRM}\|E_{IRs}$	0.986	0.929	0.957
	7	HRM+Asm2vec	$E_{HRM}\|E_{risc}$	0.986	0.949	0.964
	8	HRM+Asm2vec	$E_{HRM}\|E_{cisc}$	0.986	0.956	0.971
ref	9	HRM	E_{HRM}	0.983	0.926	0.954
	10	***Prism***	$E_{HRM}, E_{risc}, E_{cisc}$	**0.999**	**0.999**	**0.999**

Table 4: The effectiveness of HRM.

Code representation	Prec.	Recall	F1
$E_{cisc}\|E_{risc}\|E_{pca}$	0.985	0.963	0.974
$E_{cisc}\|E_{risc}\|E_{head}$	0.986	0.967	0.976
$E_{cisc}\|E_{risc}\|E_{tail}$	0.986	0.965	0.975
Prism	**0.999**	**0.999**	**0.999**

In the GR I, the assembly language outperforms IR in the clone detection task. IR is modelled after an abstract machine. The compiler's process of converting high-level language to IR and then to machine code results in a complete lack of awareness of the underlying architecture at the IR level. and thus no trace of program execution relying on hardware characteristics. Assembly language is the sequence of machine instruction execution, which describes the logic of program execution in the underlying architecture of the machine, and this sequence compares the similarity between programs in the trajectory of machine behavior. Compared with IR, assembly language is more suitable for code clone detection tasks as it provides behavior semantics to comprehend programs.

In the GR II, we combined high and low-level program languages for code representation. Experiment IDs 5-8 show that the code representation with assembly code still achieves the best results. At the same time, we find that the code representation generated by the combination of high-level and low-level languages can improve the performance of code clone detection. Moreover, the comparison of ID.5 and ID.6 with ID.9 indicates that the addition of IR has not significantly improved the detection performance due to the overlapping semantic spaces between IR and high-level languages. The comparison of ID.7 and ID.8 with ID.10 indicates that a single ISA brings different performance benefits due to their characteristics, and the combination of different ISAs can complement each other and enhance the understanding of semantics.

5.4 The Effectiveness of HRM

We conduct fine-tuning on the *HRM* to generate the E_{HRM}. To compare the effectiveness of the task-level embedding, we also implemented another three approaches for token-level embedding, namely intercepting the first 96 elements of the BERT output (E_{head}), the end 96 elements (E_{tail}) and conducting a principal components analysis (PCA) [61] the BERT output to generate a 96-dimensional vector (marked as E_{pca}). After concatenating them with the same E_{cisc} and E_{risc}, they are fed into the shallow classifier to evaluate performance. The results are shown in Tab. 4. It can be seen that *Prism* outperforms other token-level methods.

It is also notable that LAM is fine-tuned with the dataset from downstream task, which makes E_{LAM} a task-level embedding. Moreover, We visualized the comparison between the task-level embedding and token-level embedding (i.e.E_{pca}). We randomly selected two cloning tasks in the dataset, each with 50 source code files, and obtained E_{HRM} and E_{pca} for each file. As shown in Fig. 8, they are visualized using the t-SNE method, and the left and right figures show the two cloning tasks, respectively. Task-level embedding (dark blue dots) shows higher spatial aggregation. They learned knowledge from the task domain, thus helpful for the CCD.

Table 5: Comparisons of our fusion module and existing multi-feature fusion methods. *Compensated models with the same parameter numbers as our fusion module.

Method	Params	Prec. (%)	Recall (%)	F1 (%)
Addition	0.019M	79.611	74.489	76.965
Addition*	0.224M	94.230	91.789	92.993
Concatenation	0.167M	95.732	93.107	94.401
Concatenation*	0.224M	97.264	96.088	96.672
$(F_r \times F_l) * F_c$	0.224M	99.767	99.508	99.637
$(F_c \times F_l) * F_r$	0.224M	99.317	99.276	99.296
Prism	0.224M	**99.865**	**99.886**	**99.886**

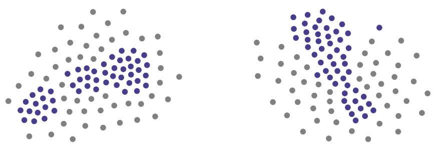


Figure 8: Visualization of task and token level embedding. Darkblue: E_{HRM} (i.e. task embedding), Gray: E_{pca} (i.e. token embedding).

5.5 Feature Fusion Module Analysis

In this section, we evaluated the performance of the feature fusion module. The results of our evaluation, as presented in Tab. 5, demonstrate the superiority of our fusion method over the addition and concatenation methods. Moreover, to ensure a fair comparison, we compensate the shallow classifier networks of baseline methods with additional FC layers. Our feature fusion modules outperform baseline methods under the unified model capacity setting. This is because addition and concatenation operations make it difficult to generate cross-index interactions. As for dealing with the feature spaces such as E_{cisc} and E_{risc}, it will hinder generating the discriminative representation, as shown in Fig. 4. *Prism* achieves high-order interactions among multiple features, which is more suitable for code representations containing assembly language.

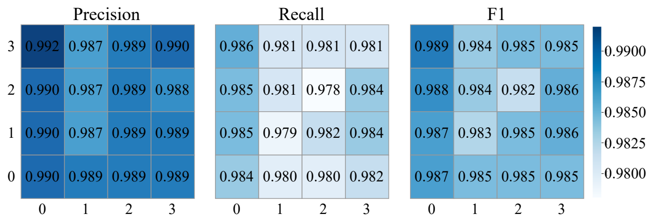


Figure 9: Compiler sensitivity analysis. The coordinate XY represents the combination of ARM-Ox and x86-Oy combination.

Further, we evaluated the effect of feature interaction order on the fusion module. As shown in Tab. 5, The $(F_r \times F_h) * F_c$ denotes the 2-D convolution kernel generated by F_{cisc} interacts with the multiplication obtained by F_{risc} and F_{hrm}, according to Eq. (3) and Eq. (4). It can be seen that the order of feature interaction has a weak impact on the *Prism*, illustrating the robustness of the *FFM*.

5.6 Sensitivity to Compiler Optimization

As the assembly code generated varies with different compiler optimization configurations, it may cause the inconsistent code statement to affect the semantic normalization result. We evaluate the sensitivity of *Prism* by changing the optimization option from O0 to O3 and compare the performance.

We first investigated the code representations obtained through concatenation, the same experiment setting with Section 5.2. Fig. 9 provides the performance variation with different optimization levels averaging the five experiments. The XY-coordinate shows the combination of optimization levels, for example, 13 means O1 for ARM and O3 for x86. Our findings suggest that diverse instruction optimizations can make it harder to align semantics when code representation fuses signal assembly code. However, we observed that combining various platforms and different priority levels had a weak impact on the clone detection task, with precision values differing by only 0.005, recall by 0.008, and F1 by 0.007.

Furthermore, we also tested the sensitivity of *Prism* to optimization levels. In all 16 experiments, Prism achieved an F1 score of 0.998 (std = 0.001), indicating that *FFM* can capture higher-order relationships between features and mitigate the impact of compiler bias. Notably, our results indicate that *Prism* has less dependence on the optimization level, making it relatively stable. We hypothesize that the instruction and memory optimization techniques exploited by *Prism* are widely used in the compiler software stack, which enables *Prism* to work well even with the preliminary optimization.

5.7 Sensitivity to Dataset

We analyze the impact of the selection dataset on Prism. we build multiple datasets comprising varying numbers of tasks randomly selected from the OJclone dataset. Subsequently, we evaluate the detection performance of Prism on these datasets. we observe similar detection performance across different groups (mean = 99.829, std = 0.089), as shown in Fig 10. This suggests that Prism exhibits low sensitivity to diverse datasets. Moreover, we find that the result may benefit from the *FEM* module. As Fig 10 shows, comparing

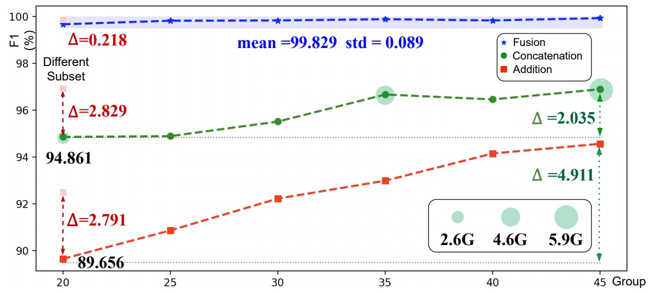


Figure 10: Dataset sensitivity analysis.

Table 6: Comparison of results from different methods on GCJ.

Method	Precision	Recall	F1
ASTNN [64] (ICSE'2019)	0.954	0.872	0.911
FCCA [25] (IEEE TR'2020)	0.967	0.898	0.931
InferCode[10] (ICSE'2021)	0.932	0.926	0.929
TreeCen [24] (ASE'2022)	0.950	0.950	0.950
HELoC [58] (ICPC'2022)	0.987	0.959	0.974
Prism	**0.999**	**0.999**	**0.999**

different feature fusion methods (addition and concatenation), we find that as the dataset size increases from 2.6G to 5.9G, the detection performance improves. While the Prism exhibits robustness to dataset variations. This demonstrates the effective higher-order interactions in *FEM* module yielding discriminative features that contribute to the clone detection. Even when considering 20 tasks, Prim achieved the best performance. Finally, we build two distinct sub-datasets, each containing 20 tasks with no overlap. We observed varying performances of different fusion methods, while Prism maintained stability and exhibited good performance. we can conclude that Prism exhibits low sensitivity to different datasets.

5.8 Cross Language Code Clone Detection

We employ the concept of *Prism* to conduct cross-lingual validation on the GCJ dataset [36], which is widely used for Java clone detection tasks, specifically for detecting semantic clones in Java code. We have also fine-tuned the *HRM* model using the GCJ dataset. However, unlike previous work, we utilize disassembled Java bytecode as a source of rich behavioral semantics, which is then used to generate semantic embedding directly with *sentencer-bert* [48]. With only two features, we fuse them using matrix multiplication as described in Eq. (3), and then generate the final code representation using aggregation and residual connections as described in Eq. (5). The resulting code representation has a dimension of 528. A total of 1,054,022 data pairs are generated and the experimental parameters are kept constant with our prior work.

Tab. 6 shows the superiority of the *Prism* on GCJ datasets. Our method improves the performance by about 4.0% in recall compared to HELoc. This observation highlights the benefits of combining syntactic and semantic information from high-level and low-level languages to enhance code representation, which is also the core idea behind the design of our proposed method *Prism*.

Table 7: Code classification accuracy (%) on GCJ and OJClone.

Method	OJClone	GCJ
ASTNN [64] (ICSE'2019)	98.2	93.4
InferCode [10] (ICSE'2021)	96.3	90.8
HELoC [58] (ICPC'2022)	99.6	97.2
Prism	**100**	**99.7**

5.9 Code Representation for Code Classification

To validate the effectiveness of the proposed code representation framework, we conduct code classification experiments on our dataset based on the code representations generated by *Prism*. We conducted experiments on the OJClone [40] and GCJ [36] datasets. For the GCJ dataset, we selected 12 classes and a total of 1625 code files. Tab. 7 demonstrates that our approach surpasses all compared baseline methods by achieving significantly higher accuracy rates on both the OJClone and GCJ datasets. It indicates that *Prism* can accurately capture the behavior semantics of the source code.

6 RELATED WORK

In code clone detection, the code representation determines the upper limit of task performance [17]. Syntactic and semantic information are significant features for code representation [17, 37].

Syntax-based approaches encode program snips, either by dividing the program into strings and tokens or parsing the program into a parse tree or abstract syntax tree (AST) [37]. DECKARD [27] and RtvNN [60] are syntax-based clone detection methods. Additionally, there are various syntax-based code representation models, including the following: Code2vec [29], Code2seq [4], CodeBERT [18], InferCode [10], C-BERT [11], ASTNN [64] and TreeCen [24]. The above clone detection methods only focus on the syntactic information of the code and ignore the semantic information. However, our method fuses syntactic and semantic information for learning.

Recent work has emphasized the significance of semantic information, combining syntax and semantics for learning. These approaches capture high-level semantic information about source code by utilizing control flow graphs, data flow graphs, or call functions in the source code. Deepsim [65], FA-AST [56], FCDetector [17] are clone detection methods based on syntactic and semantic information. There are also some code representation models based on fusion learning, including GraphCodeBERT [20] and GraphCode2Vec [37]. Dependence on high-level language to extract semantic information may be misleading in analyzing the semantics of programs. Unlike them, our approach fuses multiple languages and resolves the misleading dilemma of high-level PL.

Numerous works make use of a low-level program language to comprehend programs. Compared with high-level PL, IR is more appropriate for learning code representation because it is modeled on an abstract machine with a finite instruction set that can be well matched to operational semantics [44]. IR2Vec [55] and OSCAR [44] both capture program semantics with an LLVM IR representation of the program. Additionally, NCC [8] exploits both the underlying data flow and control flow of the program to improve the model's generalization capacity. After revisiting assembly in code representation, especially for code clone detection, we find that assembly language can also reflect underlying instruction execution information, which is not inferior to IR. To our best knowledge, the *Prism* is the first code clone detection method that uses both high-level program language and assembly code. It can involve both syntactic and underlying semantic knowledge.

7 DISCUSSION

7.1 Time performance of Prism

The time consumption of Prism can be divided into three stages. The first stage involves generating different assembly language files, the second stage involves generating the embedding for assembly and high-level languages, and the third stage involves training the model. In the first stage, we take 103 seconds to use `aarch64-linux-gnu-gcc -s` and `x86_64-linux-gnu-gcc -s` to generate assembly code files. In the second stage, we train the Asm2vec model and fine-tuned the BERT model. We use 3,500 files to train the Asm2vec model, which takes 784 seconds. We also use 7,000 code pairs to fine-tune the BERT model, which takes 16220 seconds. In the third stage, following the data processing in Sec 4.2, we obtain 700,000 data items, and the dataset size is approximately 4.6G. Finally, We use the training dataset to train the feature fusion module and complete the code clone detection task. The training time was 744 seconds. Finally, the total training time of our model was 17,851 seconds. The inference time for the testset is 0.46 seconds. Our model demonstrates favourable time performance. This efficiency can be attributed to the compiler and significant acceleration provided by modern hardware, such as GPUs.

7.2 Generalization of Prism

We validated the clone detection performance of Prism on datasets in both C and Java programming languages. However, when applying Prism to new datasets or applying it as a representation model in novel task scenarios, we recommend fine-tuning the pre-trained model to ensure stable performance.

8 CONCLUSION

In this paper, we propose *Prism*, which uses behavior semantics digging from the assembly language of multiple architectures for the first time. We revisit the assembly code in code clone detection, finding that assembly code provides the semantics of program execution behavior, a new perspective to understanding the program. Moreover, we introduce a multi-feature fusion strategy that leverages global information interaction to fusion multiple assembly instruction sets, which benefits semantic understanding. *Prism* can successfully avoid semantic misunderstanding caused by high-level language diversity. We conducted experiments on the OJClone dataset, which shows that *Prism* achieves state-of-the-art performance in code clone detection.

9 ACKNOWLEDGEMENTS

This work was supported in part by the Natural Science Foundation of China (62172239,62372254), Key Technologies R&D Program of Guangdong Province, China (2021B0101310002), Shandong Provincial Natural Science Foundation, China (ZR2022LZH009, ZR2023L ZH003) and Open Fund of PDL (WDZC20215250123).

REFERENCES

[1] 2014. copilot. https://github.com/features/copilot.

[2] 2014. Crosstool-NG. https://crosstool-ng.github.io/.

[3] 2014. Refactoring.Guru. https://refactoring.guru/extract-method.

[4] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating Sequences from Structured Representations of Code. *CoRR* abs/1808.01400 (2018). arXiv:1808.01400

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[6] Mihail Asăvoae. 2014. K Semantics for Assembly Languages: A Case Study. *Electronic Notes in Theoretical Computer Science* 304 (2014), 111–125. https://doi.org/10.1016/j.entcs.2014.05.006 Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011)..

[7] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[8] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *CoRR* (2018). arXiv:1806.07336

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[10] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1186–1197.

[11] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. 2020. Exploring Software Naturalness through Neural Language Models. *CoRR* abs/2006.12641 (2020). arXiv:2006.12641

[12] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[15] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[16] William B. Dolan and Chris Brockett. 2005. Automatically Constructing a Corpus of Sentential Paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*.

[17] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *CoRR* abs/2002.08155 (2020). arXiv:2002.08155

[19] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. 321–330.

[20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* abs/2009.08366 (2020). arXiv:2009.08366

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[22] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR* abs/1207.0580 (2012). arXiv:1207.0580

[23] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735

[24] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. 2023. TreeCen: Building Tree Graph for Scalable Semantic Code Clone Detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 109, 12 pages. https://doi.org/10.1145/3551349.3556927

[25] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2020. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability* 70, 1 (2020), 304–318.

[26] Tongwen Huang, Zhiqi Zhang, and Junlin Zhang. 2019. FiBiNET: Combining Feature Importance and Bilinear Feature Interaction for Click-through Rate Prediction. In *Proceedings of the 13th ACM Conference on Recommender Systems* (Copenhagen, Denmark) *(RecSys '19)*. Association for Computing Machinery, New York, NY, USA, 169–177. https://doi.org/10.1145/3298689.3347043

[27] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[28] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pretrained Contextual Embedding of Source Code. *CoRR* (2020). arXiv:2001.00059

[29] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. 2019. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–12.

[30] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting Working Code Examples. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 664–675. https://doi.org/10.1145/2568225.2568292

[31] Diederik Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations* (12 2014).

[32] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE, 301–309.

[33] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32* (Beijing, China) *(ICML'14)*. JMLR.org, II–1188–II–1196.

[34] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. *CoRR* abs/2104.08691 (2021). arXiv:2104.08691

[35] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, San Francisco, CA.

[36] Yuding Liang and Kenny Q. Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (New Orleans, Louisiana, USA). Article 641, 8 pages.

[37] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. Zhang, M. Papadakis, M. Cordy, X. Xie, and Y. L. Traon. 2021. GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses. *arXiv e-prints* (2021).

[38] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *International Conference on Machine Learning (ICML)*.

[39] Luiz G. A. Martins, Ricardo Nobre, João M. P. Cardoso, Alexandre C. B. Delbem, and Eduardo Marques. 2016. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Trans. Archit. Code Optim.* 13, 1, Article 8 (mar 2016), 28 pages. https://doi.org/10.1145/2883614

[40] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI conference on artificial intelligence*.

[41] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K. Roy, and Kevin A. Schneider. 2019. CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1026–1037. https://doi.org/10.1109/ASE.2019.00099

[42] Manziba Akanda Nishi and Kostadin Damevski. 2018. Scalable code clone detection and search based on adaptive prefix filtering. *Journal of Systems and Software* 137 (2018), 130–142.

[43] David A. Patterson and David R. Ditzel. 1980. The Case for the Reduced Instruction Set Computer. *SIGARCH Comput. Archit. News* 8, 6 (oct 1980), 25–33. https://doi.org/10.1145/641914.641917

[44] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs? *International Conference on Machine Learning (ICML)* abs/2105.04297 (2021). arXiv:2105.04297

[45] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *CoRR* abs/1802.05365 (2018). arXiv:1802.05365

[46] Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. 2002. Finding plagiarisms among a set of programs with JPlag. *J. Univers. Comput. Sci.* 8, 11 (2002), 1016.

[47] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100, 000+ Questions for Machine Comprehension of Text. *CoRR*

abs/1606.05250 (2016). arXiv:1606.05250

[48] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

[49] Abdullah Sheneamer. 2018. CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 701–706. https://doi.org/10.1109/ICMLA.2018.00111

[50] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019. How to Fine-Tune BERT for Text Classification?. In *Chinese Computational Linguistics*, Maosong Sun, Xuanjing Huang, Heng Ji, Zhiyuan Liu, and Yang Liu (Eds.). Springer International Publishing, Cham, 194–206.

[51] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090. https://doi.org/10.1109/TSE.2015.2448531

[52] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone Refactoring with Lambda Expressions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 60–70. https://doi.org/10.1109/ICSE.2017.14

[53] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[55] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (dec 2020), 27 pages.

[56] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[57] Wei Wang, Ming Yan, and Chen Wu. 2018. Multi-granularity hierarchical attention fusion networks for reading comprehension and question answering. *CoRR* abs/1811.11934 (2018). arXiv:1811.11934

[58] Xiao Wang, Qiong Wu, Hongyu Zhang, Chen Lyu, Xue Jiang, Zhuoran Zheng, Lei Lyu, and Songlin Hu. 2022. HELoC: Hierarchical Contrastive Learning of Source Code Representation. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 354–365. https://doi.org/10.1145/3524610.3527896

[59] Hui-Hui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code *(IJCAI'17)*. AAAI Press, 3034–3040.

[60] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *the 31st IEEE/ACM International Conference*.

[61] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[62] Lingfeng Yang, Xiang Li, Renjie Song, Borui Zhao, Juntian Tao, Shihao Zhou, Jiajun Liang, and Jian Yang. 2022. Dynamic MLP for Fine-Grained Image Classification by Leveraging Geographical and Temporal Information. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10935–10944. https://doi.org/10.1109/CVPR52688.2022.01067

[63] Yang Yuan and Yao Guo. 2011. CMCD: Count Matrix Based Code Clone Detection. In *2011 18th Asia-Pacific Software Engineering Conference*. 250–257. https://doi.org/10.1109/APSEC.2011.13

[64] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[65] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.