DECodeT5: A Lightweight and Efficient Neural Decompiler with Assembly Semantic Assistance

Li Liu, Fanghui Sun, Shen Wang, Xunzhi Jiang

***Abstract*—Decompilation plays a critical role in firmware analysis and reverse engineering by enabling the recovery of high-level source code from binary executables. However, existing neural decompilation models often face challenges due to the semantic gap between assembly code and high-level languages, and they typically require large-scale models that impose significant computational demands. In this paper, we propose DECodeT5, a lightweight and efficient neural decompilation method for the C language. DECodeT5 builds upon the CodeT5 code generation model by integrating a pre-trained assembly encoder in place of its original embedding layer. This modification allows for more effective semantic learning from assembly code, improving the model's ability to capture the intricacies of assembly-to-source code mapping. The architecture of DECodeT5 not only accelerates convergence during end-to-end decompilation tasks but also supports rapid adaptation across different compilers and instruction set architectures (ISAs) by swapping out the encoder module as needed. Our experimental evaluation on the HumanEval and Exebench benchmarks reveals that DECodeT5 significantly improves semantic recovery accuracy, outperforming Ghidra by over 83% and LLM4Decompile by more than 43% in terms of execution recovery. Furthermore, DECodeT5 maintains a compact model size of only 360M parameters, providing inference speeds that are twice as fast as LLM4Decompile. These results underscore DECodeT5's suitability for deployment in resource-constrained environments and its flexibility in adapting to real-world reverse engineering scenarios, offering a practical solution for modern firmware analysis and security tasks.**

***Index Terms*—Decompile, reverse engineering, firmware security, deep neural network.**

I. Introduction

COMPILATION is the process of converting source code written in high-level programming languages into machine instructions that can be executed by a computer. During this process, the compiler transforms high-level language code into binary machine code with equivalent execution semantics. Decompilation, as the reverse of this process, aims to recover equivalent high-level source code from compiled binary files. As a critical technique in binary analysis, decompilation is widely used in software security and reverse engineering tasks such as vulnerability detection, malware analysis, security auditing, and code migration [1]–[4].

This work is supported by the National Defense Basic Scientific Research Program of China (No. JCKY2023603C043), the Key RD Plan of Heilongjiang Province (No. 2022ZX01C01), and Natural Science Foundation of Heilongjiang Province of China (No. LH2024F023).

L. Liu, F. Sun, S. Wang, X. Jiang are with the School of Cyberspace Science, Harbin Institute of Technology, Harbin 150001, China (email: liuliqaz@stu.hit.edu.cn; sunfanghui@hit.edu.cn; shen.wang@hit.edu.cn; jiangxunzhi@hit.edu.cn).

However, recovering source code from binary is a challenging task. The compilation process discards a significant amount of semantic information—for instance, the names of variables and functions are lost, control flow structures are often lowered to basic jump instructions, and operations on variables are replaced by low-level data movement and arithmetic instructions. Moreover, variations introduced by different compiler optimizations and instruction set architectures (ISAs) lead to substantial differences in the binary outputs generated from the same source code. These factors make it difficult to accurately reconstruct the original structure and semantics of the source code during decompilation. The challenge is further amplified in IoT devices, where firmware is often compiled with aggressive optimizations and cross-compiled across diverse ISAs, adding additional complexity to the decompilation process [5].

Traditional decompilation tools (such as Ghidra [6] and IDA Pro [7]) typically rely on rule-based approaches for code recovery. They identify variable information using predefined rules and reconstruct control structures in the source code through subgraph matching strategies on control flow graphs. These methods often struggle with complex or unknown control flow structures.

In recent years, an increasing number of studies have explored the use of artificial neural networks to learn the relationship between low-level assembly instructions and high-level programming languages. A typical approach is to treat the decompilation process as a translation task between two languages, applying sequence-to-sequence models from neural machine translation (NMT) to learn the semantic mapping from assembly code to source code [8], [9]. More recently, some works have leveraged large language models (LLMs) to approach decompilation as a code generation task, using stacked decoders to generate more accurate and readable high-level code. Despite these advancements, current decompilation methods still face several limitations.

First, rule-based approaches are not well-suited for handling unknown or optimized control flow structures, often generating excessive GOTO statements that reduce code readability. Although some studies [10], [11] attempt to alleviate this issue using compiler optimization knowledge, this significantly increases the complexity of rule design and the cost of cross-platform adaptation, making it difficult to cope with the diverse compilation environments typical of firmware reverse engineering.

Second, the generation quality of neural decompilation methods often depends on large-scale models and their code understanding capabilities. While large language models [12],

[13] have achieved state-of-the-art performance in decompilation, their parameter sizes often reach billions, resulting in extremely high training and inference costs, making them impractical for deployment in resource-constrained environments and unsuitable for efficient adaptation to the diverse instruction set architectures commonly found in firmware.

Therefore, designing a neural decompilation method that can efficiently operate under limited computational resources while maintaining good readability and transferability across different compilation conditions (compilers and instruction set architectures) is of significant importance for reducing tool development costs and improving practical applicability.

In this paper, we propose DECodeT5, a function-level C language decompilation method based on Transformer [14]. DECodeT5 adopts a modular design that integrates assembly semantics into an existing code generation model, enabling high-quality C code generation and rapid adaptation across different compilers and instruction set architectures. Moreover, with a model size of only 360M parameters, DECodeT5 remains efficient even in resource-constrained environments.

In this paper, we propose DECodeT5, a neural decompilation framework for function-level C code recovery from assembly instructions. Unlike prior work that directly applies general-purpose sequence-to-sequence models, DECodeT5 introduces a domain-adapted architecture that explicitly bridges the semantic gap between low-level assembly and high-level source code. The proposed method is designed to learn a robust cross-level semantic mapping while maintaining practical efficiency, achieving competitive performance with only 360M parameters.

The backbone of DECodeT5 is a modular hybrid Transformer [14] architecture constructed around a pretrained assembly encoder and a CodeT5-based [15] code generation backbone. Rather than using CodeT5 in an off-the-shelf manner, we replace its original embedding layer with the specialized assembly encoder, which injects instruction semantics, operand structures, and control-flow patterns directly into the model's latent space. This structural modification fundamentally changes the model's input representation and equips the framework with task-specific inductive bias for neural decompilation.

To further align the heterogeneous semantic spaces of assembly and source code, we introduce an additional projection layer that performs explicit representation alignment between the assembly encoder and the code generation backbone. This design allows the model to preserve the strengths of large-scale denoising pretraining on source code while grounding the generation process in low-level semantics. As a result, DECodeT5 can more effectively learn the semantic correspondence between assembly and C code in an end-to-end fashion. In addition, we design a domain-aware tokenizer that extends the original CodeT5 tokenization scheme with assembly-specific symbols, including registers and memory addressing patterns. This reduces token fragmentation and semantic noise caused by naive tokenization, shortens effective sequence lengths, and improves the stability of cross-modal semantic alignment.

We have implemented a prototype of DECodeT5 using the GCC compiler under the x86-64 architecture and evaluated its performance on the HumanEval [12] and ExeBench [16] benchmark datasets. We conduct comparative experiments against the current state-of-the-art large language model-based decompilation method, LLM4Decompile, as well as the widely used decompilation tool Ghidra. Experimental results demonstrate that DECodeT5 achieves superior decompilation performance and higher efficiency compared to existing approaches. Furthermore, we conduct additional experiments under the ARM instruction set architecture and the CLANG compiler to assess the model's adaptability across different compilation environments. These results further validate the robustness and transferability of DECodeT5.

In summary, our contributions are as follows:

- We propose DECodeT5, a novel, modular neural decompilation framework that incorporates a pre-trained assembly encoder and explicit semantic space alignment into a Transformer-based code generation backbone. Furthermore, we design a domain-aware assembly tokenizer that reduces semantic noise and improves the quality of assembly representations for neural decompilation.
- DECodeT5 outperforms LLM-based approaches in code recovery quality, while offering a smaller model size and faster inference speed, making it suitable for deployment in resource-constrained environments.
- We conduct experiments across different compilers and instruction set architectures, demonstrating DECodeT5's strong transferability.
- We release the code of DECodeT5 to support further research (https://github.com/llbcsd/decodet5).

II. Motivation

In this section, we present real-world decompilation examples to illustrate the limitations of existing methods and introduce our proposed solution.

A. Challenges of tranditional decompilation

Traditional decompilation approaches primarily rely on rule-based pattern matching, which depends heavily on the internal mechanisms of specific compilers and extensive domain knowledge. However, different compilation settings—such as compiler type and optimization level—can significantly affect the resulting binary code, making it difficult for rule-based methods to generate consistent and unified source code across environments. As shown in Fig. 1(b) and (c), when decompiling x86-64 binaries compiled with GCC under O0 and O2 optimization levels, Ghidra generates noticeably different source code.

Moreover, the pattern-matching capabilities of traditional methods are limited and often fail to handle unknown or complex control flow patterns. In Fig. 1(b), Ghidra fails to fully recover certain variable types: function parameters are incorrectly inferred as `long`, and return values are marked as `undefined8`. In addition to type mismatches, unmatched code structures are often represented as GOTO statements, increasing code complexity and reducing readability. For instance, under O2 optimization (Fig. 1(b)), Ghidra cannot

```c
int func0(const char *brackets) {
    int level = 0;
    int i = 0;
    while (brackets[i]) {
        if (brackets[i] == '<') level++;
        if (brackets[i] == '>') level--;
        if (level < 0) return -1;
        i++;
    }
    if (level != 0) return -1;
    return 1;
}
```

(a) Original source code

```c
undefined8 func0(long param_1) {
  undefined8 uVar1;
  int local_10;
  int local_c;
  local_10 = 0;
  local_c = 0;
  while( true ) {
    if (*(char *)(param_1 + local_c) == '\0') {
      if (local_10 == 0) {
        uVar1 = 1;
      }
      else {
        uVar1 = 0xffffffff;
      }
      return uVar1;
    }
    if (*(char *)(param_1 + local_c) == '<') {
      local_10 = local_10 + 1;
    }
    if (*(char *)(param_1 + local_c) == '>') {
      local_10 = local_10 + -1;
    }
    if (local_10 < 0) break;
    local_c = local_c + 1;
  }
  return 0xffffffff;
}
```

(b) Ghidra-11.3.1 (GCC O0)

```c
int func0(char *param_1) {
  char cVar1;
  int iVar2;
  bool bVar3;
  cVar1 = *param_1;
  if (cVar1 == '\0') {
    iVar2 = 1;
  }
  else {
    param_1 = param_1 + 1;
    iVar2 = 0;
    do {
      while (cVar1 != '<') {
        if ((cVar1 == '>')&&(bVar3 = iVar2 == 0,
            iVar2 = iVar2 + -1, bVar3)) {
          return iVar2;
        }
        cVar1 = *param_1;
        param_1 = param_1 + 1;
        if (cVar1 == '\0') goto LAB_00100040;
      }
      cVar1 = *param_1;
      param_1 = param_1 + 1;
      iVar2 = iVar2 + 1;
    } while (cVar1 != '\0');
LAB_00100040:
    iVar2 = (-(uint)(iVar2 == 0) & 2) - 1;
  }
  return iVar2;
}
```

(c) Ghidra-11.3.1 (GCC O2)

```c
int func0(const char *str) {
    if (*str == '\0') {
        return 1;
    }
    str++;
    int depth = 0;

    while (1) {
        unsigned char ch = *str;
        if (ch == '\0') break;
        if (ch == '<') {
            str++;
            depth++;
            ch = *str;
            if (ch == '\0') break;
            continue;
        }
        if (ch == '>') {
            depth--;
            if (depth < 0) return -1;
        }
        str++;
    }
    if (depth == 1) {
        return 0;
    } else {
        return -1;
    }
}
```

(d) ChatGPT-4o (GCC O2)

Fig. 1. Comparison of decompilation results by Ghidra and ChatGPT-4 under different compilation conditions. (a) Original source code. (b) Ghidra-11.3.1's decompilation result from binary compiled with GCC at optimization level O0. (c) Ghidra-11.3.1's decompilation result from binary compiled with GCC at optimization level O2. (d) ChatGPT-4o's decompilation result from binary compiled with GCC at optimization level O2.

accurately reconstruct the loop exit logic from Fig. 1(a), instead using a GOTO statement for early termination, which hinders understanding.

Additionally, such rule-based systems require constant updates to support new compilers or instruction sets, leading to increased development and maintenance costs.

B. Challenges of Neural Decompilation

Neural decompilation methods address some of these shortcomings by improving consistency and readability. These methods can automatically learn richer semantic mappings between assembly and source code from large-scale datasets. While neural machine translation-based approaches have achieved promising results, they often lack deep semantic understanding of programs and require additional structures like Intermediate Representations (IR) [17] or Abstract Syntax Trees (AST) [18] to enhance code generation.

Recently, large language models (LLMs) have demonstrated impressive performance in code understanding and generation. However, without task-specific training for decompilation, they still struggle to capture fine-grained program details. As shown in Fig. 1(d), ChatGPT-4o successfully reconstructs the loop logic within the function but generates an entirely different return logic and return value compared to the original. These models lack knowledge of compiler conventions and assembly semantics, and rely solely on their general code generation abilities, which limits their accuracy in function recovery. Furthermore, the massive scale of these models—often with billions of parameters—leads to high training and deployment costs, limiting their applicability in resource-constrained environments.

In fact, the relationship between assembly code and high-level source code is fundamentally different from that between natural languages. Natural languages often have synonymous vocabulary and similar grammatical structures, which facilitate semantic alignment and translation. In contrast, assembly code represents low-level machine instructions that implement high-level language constructs, typically lacking explicit semantic equivalence. As a result, the core challenge of neural decompilation lies in accurately understanding the execution semantics conveyed by the assembly code and transforming it into semantically equivalent high-level code.

To address this challenge, we propose integrating a pre-trained assembly encoder into a code generation model by replacing its original encoder embedding layer. This design leverages the strengths of both components: the code generation model's ability to produce high-quality high-level code, and the assembly encoder's capacity to comprehend low-level semantics. Together, they enable more accurate and efficient decompilation. Moreover, our approach adopts a lightweight model architecture with significantly fewer parameters compared to large language models, allowing for efficient training and deployment in resource-constrained environments. The modular design also provides strong scalability, enabling the model to be adapted across different instruction set architectures and compiler configurations.

III. Methodology

A. Overview

DECodeT5 consists of an assembly encoder and a code generation model. The assembly encoder serves as the encoding module, responsible for extracting semantic information from function-level assembly code and generating corresponding embeddings. The CodeT5-based code generation model acts as the decoding module, receiving the semantic vectors and producing high-quality source code semantically equivalent to the input assembly code. As illustrated in Fig. 2, the overall workflow of DECodeT5 comprises three main phases:

Data Preprocessing: We extract individual functions from a collected source code dataset, compile them into binary machine code, and then disassemble them into assembly code. This assembly code is further normalized and paired with the corresponding source code for training.

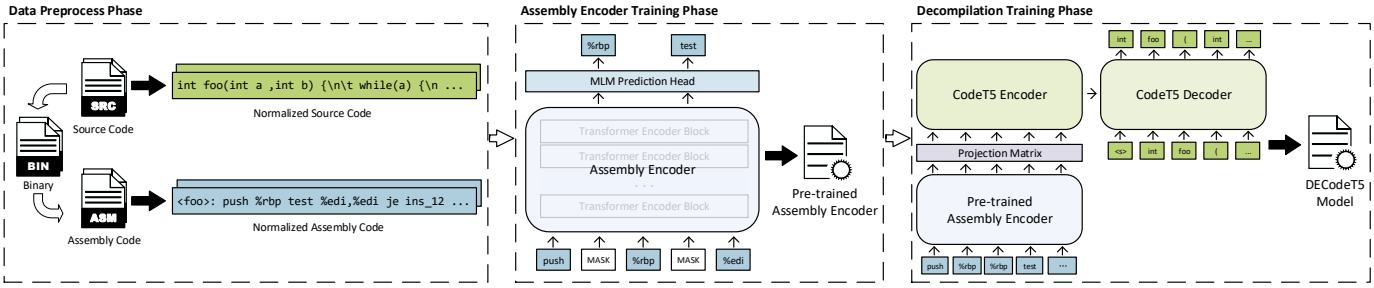


Fig. 2. The overall workflow of DECodeT5

Assembly Encoder Pre-training: A Transformer encoder is trained on the normalized assembly code using a masked language model (MLM) task. This self-supervised learning process enables the encoder to capture semantic features from the assembly code.

Decompilation Training: The CodeT5 model, originally designed for code and natural language generation, is used as the decoding module. We replace its original embedding layer with the pre-trained assembly encoder and add a projection matrix. During decompilation training, the model is fine-tuned end-to-end using the paired assembly-source code data. With the support of the assembly encoder, CodeT5 effectively learns the semantic mapping from assembly instructions to high-level source code.

B. Data Preprocessing

During data preprocessing, we aim to obtain normalized source code and normalized assembly code. Since the source code dataset is organized at the function level, we compile each individual function using GCC under x86-64 architecture to generate an object file containing only the target function. We then use the native GNU disassembler, OBJDUMP, to disassemble the object file and extract the assembly code for each C function. Next, we perform normalization on both the extracted source code and assembly code. This step removes irrelevant or noisy information from both the C functions and the assembly instructions, resulting in clean, standardized code sequences that can be directly used as input to the model.

1) Assembly code normalization: We normalize the disassembled output generated by OBJDUMP to remove unnecessary metadata, restore missing function call information, and enrich the code with explicit control flow details. Our disassembly code normalization process is shown in Fig. 3.

Because the source code is compiled without linking, the disassembled assembly lacks valid target addresses for external function calls. As shown in Fig. 3(a), all call instructions have empty operands, and the disassembler fills these with the address of the next instruction. To recover this information, we parse the .rela.text section in the ELF file header, locate the byte offset of each external call, and restore the corresponding function names as operands for the call instructions.

To further enhance the representation of control flow and eliminate noise from absolute addresses, we normalize the operands of all jump instructions. Specifically, we replace hexadecimal instruction addresses with a unified format: `ins_index`, where `index` refers to the target instruction's index in the sequence. For instance, as shown in Fig. 3(b), the instruction `je 2c` at address `7` is normalized to `je ins_14`.

Additionally, to facilitate tokenization, we replace all tabs with spaces and separate every instruction with a space. We also prepend the function name to the assembly sequence, producing a fully normalized representation suitable for input into the model.

2) Source code normalization: After compilation, source code loses its original comments and structural formatting, and these elements are not preserved in the assembly code. Recovering such high-level information directly from assembly code is challenging. Therefore, during source code normalization, we remove redundant elements that are imperceptible to the assembly code and standardize the code structure. The details of normalization process are as follows:

- Remove all inline comments and compiler preprocessing directives from the function.
- Strip away function modifiers to ensure all function headers follow a consistent format: `<return type> <function name>(<parameter list>)`.
- Eliminate all blank lines and reformat the code using clang-format to enforce a uniform coding style across all functions.

Finally, we retain line breaks and indentation (represented by tabs) to preserve the structural layout, resulting in a normalized source code sequence suitable for training.

C. Assembly Encoder Training

To enhance assembly semantic representation during end-to-end decompilation learning, we introduce an assembly encoder to generate semantic embeddings from assembly code. As shown in Fig. 2, the encoder is built upon a backbone network composed of stacked Transformer Encoder Blocks (TEBs). During the assembly encoder training phase, the encoder first tokenizes normalized assembly code using a custom assembly tokenizer. It then learns semantic representations through a masked language model (MLM) task.

1) Domain-aware Assembly Tokenizer: Our approach adopts CodeT5 as the source code generation module for decompilation. CodeT5 uses a Byte-level BPE (BBPE) tokenizer [19], which is well-suited for tokenizing high-level programming languages and natural language. However, since this tokenizer is originally trained on source code and natural

```
0000000000000000 <foo>:
 0:   f3 0f 1e fa           endbr64
 4:   53                    push   %rbx
 5:   85 ff                 test   %edi,%edi
 7:   74 23                 je     2c <foo+0x2c>
 9:   85 f6                 test   %esi,%esi
 b:   75 13                 jne    20 <foo+0x20>
 d:   89 fb                 mov    %edi,%ebx
 f:   48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi # 16 <foo+0x16>
16:   e8 00 00 00 00        callq  1b <foo+0x1b>
1b:   83 fb 01              cmp    $0x1,%ebx
1e:   74 0c                 je     2c <foo+0x2c>
20:   48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi # 27 <foo+0x27>
27:   e8 00 00 00 00        callq  2c <foo+0x2c>
2c:   48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi # 33 <foo+0x33>
33:   e8 00 00 00 00        callq  38 <foo+0x38>
38:   31 c0                 xor    %eax,%eax
3a:   5b                    pop    %rbx
3b:   c3                    retq
```

(a) Original Assembly Code

```
<foo>:
endbr64
push  %rbx
test  %edi,%edi
je    ins_14
test  %esi,%esi
jne   ins_12
mov   %edi,%ebx
lea   0x0(%rip),%rdi
callq puts
cmp   $0x1,%ebx
je    ins_14
lea   0x0(%rip),%rdi
callq puts
lea   0x0(%rip),%rdi
callq puts
xor   %eax,%eax
pop   %rbx
retq
```

(b) Normalized Assembly Code

Fig. 3. Example of assembly code normalization.

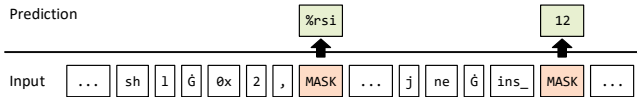


Fig. 4. MLM pre-training task of the assembly encoder.

language data, it struggles to correctly parse certain special identifiers in assembly code.

For instance, under the x86-64 architecture, the assembly instruction `shl 0x2,%rax` is tokenized by CodeT5 tokenizer into 10 separate tokens: 'sh', 'l', 'Ġ', '0', 'x', '2', ',', '%', 'ra', and 'x'. The opcode `shl` (Shift Left)—a mnemonic derived from natural language—is segmented into the prefix 'sh' and the initial letter 'l', corresponding to "shift" and "left". However, the hexadecimal value `0x2` and register `%rax` are fragmented into multiple tokens, making it difficult for the model to capture their semantics.

To enhance the model's ability to parse assembly code while preserving compatibility with the CodeT5 semantic space, we extend the original tokenizer by incorporating domain-specific tokens from assembly code. Specifically, we add all register names, the hexadecimal prefix `0x`, and the jump target prefix `ins_` as complete tokens into CodeT5's vocabulary, forming a domain-aware assembly tokenizer. Under this tokenizer, the instruction `shl 0x2,%rax` is tokenized into 7 tokens: 'sh', 'l', 'Ġ', '0x', '2', ',', and '%rax', significantly reducing unnecessary splits.

This domain-aware assembly tokenizer ensures consistency in tokenization between assembly instructions, numeric values, and function names across both the assembly encoder and the CodeT5 decoder, thereby aligning their semantic spaces. It improves the model's understanding of assembly semantics and establishes a more precise mapping between assembly and high-level code. Furthermore, by enabling dedicated representations for special assembly identifiers, the tokenizer improves expressiveness and reduces input sequence length by 20–30%.

2) Assembly Encoder Training Task: We train the assembly encoder using the masked language model (MLM) task to learn semantic representations from assembly code. Inspired by BERT's [20] pre-training strategy, this approach applies a cloze-style objective, enabling the encoder's backbone—a multi-layer Transformer with bidirectional attention—to capture meaningful semantic features within the assembly sequence.

Specifically, we randomly select 15% of the tokens from each normalized assembly sequence and replace them with a special MASK token. During training, the model is tasked with predicting the masked tokens based on surrounding context. The output representations of these positions are passed through a Softmax-based classification head, referred to as the MLM prediction head. As shown in Fig. 4, in an x86-64 assembly sequence, tokens like the register `%rsi` and the jump target `12` are replaced with MASK, and the MLM objective is to recover these tokens using contextual information.

Given an input assembly sequence $\mathbf{F} = [x_1, x_2, ..., x_n]$ and a set of masked positions Mask $= \{m_1, m_2, .., m_n\}$, the MLM training objective is defined as:

$$\mathcal{L}_{MLM}(\theta) = -log \prod_{m_i \in \text{Mask}} p(\hat{y_{m_i}}|\mathbf{F}), \hat{y_{m_i}} \in V_{asm} \quad (1)$$

where $\hat{y_{m_i}}$ is the predicted token at the masked position m_i produced by the Softmax classifier, and V_{asm} denotes the vocabulary of the assembly tokenizer.

By leveraging bidirectional attention to predict masked tokens, the encoder effectively learns rich semantic features from the assembly code. Moreover, since jump addresses in the assembly are normalized into target instruction indices during preprocessing, the MLM task also implicitly helps the model capture control flow semantics—further enhancing its understanding of low-level execution logic.

D. DECodeT5 Decompilation Training

After training the assembly encoder, we obtain a model AE capable of generating semantic embeddings from assembly code. We then integrate this pre-trained encoder into the CodeT5 architecture by replacing its original embedding layer—initially designed for source code and natural language—with our assembly encoder AE. This enables the model to accept assembly code as input and produce meaningful embeddings, thereby constructing the complete DECodeT5 framework as shown in Fig. 5. The following sections detail

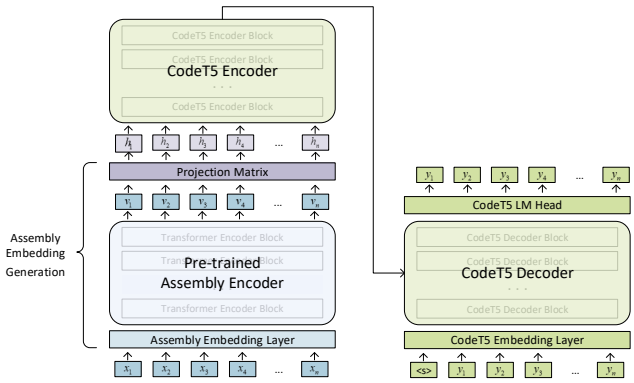


Fig. 5. The structure of DECodeT5

the process of generating assembly embeddings and the training procedure of DECodeT5.

1) Assembly Embedding Generation: Given an assembly code sequence $\mathbf{x} = [x_1, x_2, ..., x_n]$, we use the output of the final layer of the assembly encoder as the intermediate vector representation for each token: $AE(\mathbf{x}) = [v_1, v_2, ..., v_n], v_i \in \mathbb{R}^{d_h}$.

To further align these token representations with the semantic space of CodeT5, we introduce a projection matrix $W_{proj} \in \mathbb{R}^{d_h \times d_h}$. By applying matrix multiplication, we obtain the projected embedding representation of the assembly code: $H_{asm} = [h_1, h_2, ..., h_n], h_i = v_i W_{proj}, h_i \in \mathbb{R}^{d_h}$. The projection matrix is introduced to align the latent space of the pretrained assembly encoder with the semantic space of CodeT5's encoder. This linear transformation allows us to preserve assembly-specific semantics while ensuring compatibility with CodeT5's attention layers.

Finally, the projected embeddings H_{asm} are used as the input embeddings to the CodeT5 encoder, along with the attention matrix derived from the assembly sequence. The corresponding source code sequence is then processed by the standard embedding and decoding layers of the CodeT5 decoder.

2) Sequence-to-sequence Training: We formulate neural decompilation in DECodeT5 as a sequence-to-sequence learning task, aiming to model the mapping from assembly code to source code. During training, given an assembly sequence $\mathbf{x} = [x_1, x_2, ..., x_n]$ and its corresponding source code sequence $\mathbf{y} = [y_1, y_2, ..., y_m]$, the assembly encoder encodes $\mathbf{x}$ into assembly embeddings H_{asm}, which are then passed to CodeT5 to generate the source code.

The CodeT5 decoder is trained in an auto-regressive manner: the source sequence $\mathbf{y}$ is shifted one position to the right and prepended with a special start token <s>, which serves as the decoder's input to predict the next token at each step. The original source sequence $\mathbf{y}$ is used as the prediction target. The probability of generating token y_t at position t is defined as: $P(y_t \mid y_{<s>}, y_1, ..., y_{t-1}, x_1, x_2, ...x_n)$.

We use the cross-entropy loss to optimize decompilation performance. The loss function is defined as:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{m} logP(y_t^{(i)} | y_{<t}^{(i)}, \mathbf{x}^{(i)}) \quad (2)$$

where N is the number of training samples, m is the length of the source code sequence, and $P(y_t^{(i)} | y_{<t}^{(i)}, \mathbf{x}^{(i)})$ is the predicted probability of token y_t in the i-th sample.

By minimizing the difference between the predicted and true source code tokens during training, DECodeT5 learns the sequence-level mapping from assembly to source code, thereby achieving neural decompilation.

IV. Experimental Setup

DECodeT5 is implemented using the Hugging Face Transformers library [21], based on PyTorch 1.12.1 with CUDA 11.3 support. The experiments are conducted on a Ubuntu 22.04 server with 2 Intel Xeon 4216 CPUs, 256GB DDR4 RAM, and 2 NVIDIA RTX 3090 GPUs.

A. Training Dataset

We construct two separate datasets for training: an assembly code dataset for pre-training the assembly encoder, and an assembly–source code pair dataset for training the DECodeT5.

For the assembly code dataset, we compile 1.0 million C functions collected from 146 real-world projects in the AnghaBench [22]. Compilation is performed using GCC-11.4.0 under four optimization levels (O0, O1, O2, and O3) in x86-64 architecture. The resulting binary object files are disassembled using OBJDUMP. After duplicate removal and preprocessing, we obtain a clean assembly sequence dataset containing 3.2 million function-level assembly samples.

For the decompilation dataset, we use compilable C functions from real-world software provided in ExeBench [16] as source data. These functions are compiled using GCC-11.4.0 under three optimization levels (O0 to O2) in x86-64 architecture. To ensure alignment between the source and assembly code, we filter out functions containing macro definitions or inline assembly, as well as empty or duplicate functions. The compiled binaries are disassembled using OBJDUMP, and the resulting assembly is paired with the corresponding source code after preprocessing. The final decompilation dataset contains 1.8 million assembly–source code pairs.

It is important to note that this work focuses on function-level neural decompilation, with the objective of learning a robust mapping from assembly instruction sequences to localized high-level semantic structures. Therefore, neither the training datasets nor the evaluation benchmarks are compiled with the -flto option. Whether the binaries are linked primarily affects symbol resolution and address relocation but does not alter the execution semantics of already compiled functions. Since the functions in the training corpus originate from real projects as isolated, self-contained units with only external declarations added for successful compilation, directly compiling them into object files provides an efficient and scalable approach for constructing large assembly–source datasets.

B. Model Setup and Training Details

Considering experimental environment constraints and the trade-off between efficiency and performance, we design DECodeT5's assembly encoder as a 12-layer Transformer encoder with 12 attention heads, a hidden embedding dimension of 768 ($d_h = 768$), and a maximum sequence length of 512 ($n = 512, m = 512$). To mitigate the limitations of absolute positional encoding on long-sequence inference, we employ relative positional encoding. For the decompilation generation module, DECodeT5 adopts CodeT5-base (220M parameters) as its backbone. Overall, the DECodeT5 model has approximately 360M parameters, balancing capability and computational efficiency.

The assembly encoder is trained for 10 epochs (78 hours) with a batch size of 128 and a learning rate of $5e-5$. For the subsequent decompilation training, DECodeT5 is trained for 130,095 steps (6 days) with a batch size of 32 and the same learning rate ($5e-5$).

C. Evaluation Benchmarks and Metrics

1) Benchmarks: To evaluate the performance of DECodeT5, we conduct experiments on two benchmark datasets: HumanEval [12] and ExeBench-test-real [16]. HumanEval consists of 164 manually written C functions covering a range of programming tasks, including arithmetic operations, array traversal, string manipulation, and logical expressions. Each function is supplemented with necessary dependencies to ensure it can be compiled independently and is accompanied by input-output test cases and corresponding assertion checks. We also use the ExeBench-test-real dataset as an additional evaluation benchmark. This dataset contains 2,232 C functions extracted from real-world projects. Each function includes its required header files and associated I/O pairs to guarantee that it is both compilable and executable. Both datasets are compiled and linked into executable files under different optimization levels (O0 to O3). We apply the same disassembly and preprocessing procedures as used for our training data to ensure consistency in the evaluation process.

2) Metrics: We evaluate DECodeT5 in terms of its ability to recover functional semantics during decompilation and the quality of the generated code.

To validate the recovery of functional semantics during the decompilation process, we assess whether the decompiled functions exhibit the same input-output behavior as the original functions. We compile the decompiled functions by supplementing them with the necessary header files and calculate the re-compilability rate. The formula for calculating the re-compilability rate is as follows:

$$\text{re-compilability rate} = \frac{1}{N}\sum_{i=1}^{N} C(f_i) \tag{3}$$

$$C(f) = \begin{cases} 1 & \text{if function } f \text{ can be compiled} \\ 0 & \text{else} \end{cases} \tag{4}$$

If the decompiled result compiles correctly, we further use the I/O test cases provided in the benchmark to verify the execution semantics of the decompiled function and calculate the re-executability rate on the benchmark. The formula for calculating the re-executability rate is:

$$\text{re-executability rate} = \frac{1}{N}\sum_{i=1}^{N} I(f_i) \tag{5}$$

$$I(f) = \begin{cases} 1 & \text{if function } f \text{ passes all IO test} \\ 0 & \text{else} \end{cases} \tag{6}$$

To evaluate the code readability and quality of the generated functions, we calculate the edit distance between the decompiled code and the original source code. Since function and variable names in the benchmark have been anonymized and do not contain special identifiers, edit distance serves as a reliable metric for assessing the model's ability to recover the structure and content of the original code.

We adopt the edit similarity score from [9] to quantify code quality, defined as:

$$\text{Edit Similarity} = max(1 - \frac{\text{Edit Distance}}{\text{Sequence Length}}, 0) \tag{7}$$

where Sequence Length refers to the length of the target source code.

D. Baselines

We compare DECodeT5 with four baseline methods, each representing a different category of decompilation approaches: traditional decompilers, neural translation-based models, and large language model (LLM)-based methods.

Ghidra [6]: This method is a widely used traditional reverse engineering tool. We install Ghidra 11.3.1 and use the Pyhidra library to automate the decompilation of binary executables from our evaluation benchmarks.

IDA Pro [7]: This method is a widely used commercial reverse engineering tool. We use IDA Pro 7.5 along with custom IDA scripts to decompile the binary target files in the evaluation benchmark.

SLaDe [9]: This method implements an end-to-end decompilation model based on BART [23]. After decompilation, it applies PsycheC for type inference. As SLaDe directly processes assembly code generated from compilation, we re-implement SLaDe using the hyperparameters from the original paper and train it on our dataset for a fair comparison.

LLM4Decompile [12]: This method is a large-model-based decompiler built on DeepSeek-Coder [24]. It undergoes a two-stage training process on both synthetic and real-world data, achieving strong performance. We evaluate it using the officially released 1.3B parameter model.

DeepSeek-v3 [25]: To assess the limitations of general-purpose large language models in decompilation, we experiment with DeepSeek-v3, one of the most popular LLMs. We use the following prompt: "In the following conversation, please decompile the assembly code into the corresponding C function without generating additional explanations."

TABLE I
COMPARISON OF RE-COMPILABILITY RATES AND RE-EXECUTABILITY RATES OF DIFFERENT DECOMPILATION METHODS ON HUMANEVAL.

	Re-compilability Rate					Re-executability Rate				
	O0	O1	O2	O3	Average	O0	O1	O2	O3	Average
Ghidra	0.3598	0.3598	0.3780	0.3354	0.3582	0.3110	0.2744	0.2622	0.2256	0.2683
IDA pro	0.3902	0.4024	0.3963	0.3476	0.3841	0.3232	0.3232	0.3049	0.2805	0.3079
SLaDe	0.8247	0.6786	0.7554	0.7982	0.7642	0.5123	0.2012	0.1957	0.1751	0.2710
LLM4Decompile-1.3B	0.9381	0.9429	0.9065	0.9035	0.9227	0.6082	0.2786	0.2230	0.2632	0.3432
LLama3.2-1B	0.9074	0.8631	0.8525	0.8590	0.8705	0.4177	0.3647	0.1957	0.1918	0.2924
Qwen2.5-0.5B	0.8443	0.8325	0.8287	0.8189	0.8311	0.3763	0.3852	0.1891	0.1751	0.2814
Qwen2.5-coder-1.5B	0.9456	0.9429	0.9287	0.9123	0.9324	0.6541	0.2687	0.2752	0.2428	0.3602
DeepSeekV3	**0.9878**	**0.9695**	**0.9817**	**0.9207**	**0.9649**	0.5793	0.3293	0.3320	0.3015	0.3855
DECodeT5	0.9794	0.9071	0.8993	0.8947	0.9201	**0.7835**	**0.4643**	**0.3813**	**0.3412**	**0.4925**

TABLE II
COMPARISON OF RE-COMPILABILITY RATES AND RE-EXECUTABILITY RATES OF DIFFERENT DECOMPILATION METHODS ON EXEBENCH-TEST-REAL.

	Re-compilability Rate					Re-executability Rate				
	O0	O1	O2	O3	Average	O0	O1	O2	O3	Average
Ghidra	0.3119	0.3825	0.3859	0.3366	0.3512	0.1502	0.1588	0.1382	0.1411	0.1470
IDA pro	0.5534	0.4707	0.4627	0.3726	0.4648	0.1866	0.1630	0.1636	0.1521	0.1633
SLaDe	0.5019	0.4885	0.4257	0.3653	0.4453	0.2921	0.1450	0.1283	0.1107	0.1690
LLM4Decompile-1.3B	0.5361	0.5001	0.4914	0.4834	0.5027	0.3257	0.2058	0.1900	0.1955	0.2292
LLama3.2-1B	0.7391	0.8056	0.8028	0.7890	0.7841	0.1732	0.2848	0.2150	0.2030	0.2190
Qwen2.5-0.5B	0.6749	0.7861	0.7838	0.7517	0.7491	0.1279	0.2418	0.1854	0.1941	0.1873
Qwen2.5-coder-1.5B	0.8761	0.8585	0.8502	0.8412	0.8565	0.4513	0.3804	0.3560	0.3030	0.3726
DeepSeekV3	**0.9174**	**0.8790**	**0.8799**	0.8303	**0.8766**	0.4987	0.3023	0.3320	0.3242	0.3643
DECodeT5	0.9135	0.8662	0.8638	**0.8603**	0.8759	**0.6887**	**0.4836**	**0.4547**	**0.4289**	**0.5139**

V. EVALUATION

In this section, we conduct a series of experiments to compare DECodeT5 with other baseline methods, aiming to answer the following research questions:

- **RQ1:** How effectively does DECodeT5 recover the original execution semantics?
- **RQ2:** How accurate and readable is the code generated by DECodeT5?
- **RQ3:** How does DECodeT5 perform under different compilation conditions?
- **RQ4:** How fast is DECodeT5 compared to other methods?
- **RQ5:** How does DECodeT5 perform under different input length ranges?
- **RQ6:** How beneficial is the use of the assembly encoder and code generation model?

A. Comparison of Program Semantic Recovery Performance

1) Comparison with Decompilation Baselines: We evaluate the function-level semantic recovery capabilities of various decompilation methods on two benchmark datasets: HumanEval and Exebench-test-real. Considering that the maximum sequence length for DECodeT5 during training is 512, we filter out overlength samples from the validation sets.

Table I and Table II present the re-compilability rate and re-executability rate of each method across two datasets and four different compilation optimization settings under x86-64 architecture. As shown, DECodeT5 demonstrates excellent decompilation performance across all evaluations. Most of the decompiled results on both HumanEval and Exebench-test-real can be successfully compiled, with average re-compilability rates of 0.9201 and 0.8759, respectively. Furthermore, DECodeT5 achieves average re-executability rates of 0.4925 and 0.5139, significantly outperforming other baseline methods. Notably, on Exebench-test-real, DECodeT5's average re-executability rate is 3.5 times higher than Ghidra (0.1470), 3 times higher than SLaDe (0.1690), 40% higher than DeepSeek-v3 (0.3643), and 2.2 times higher than LLM4Decompile-1.3B (0.2292).

The rule-based decompilation method Ghidra achieves the lowest average re-compilability and re-executability rates, with values of only 0.3582 and 0.2683 on HumanEval. Although Ghidra's rule-based decompiled results align with the machine behavior of assembly instructions, it produces numerous variables whose types cannot be identified, making it difficult for most decompiled results to compile correctly. However, among the decompiled outputs that do compile successfully, approximately 75% pass the I/O tests. This indicates that while Ghidra's output may lack syntactic completeness for compilation, it often retains accurate functional semantics. IDA Pro outperforms Ghidra in the HumanEval experiments, achieving an average re-compilability rate of 0.3841 and an average re-executability rate of 0.3079.

The LLM4Decompile series, fine-tuned from DeepSeek-Coder, also performs well on HumanEval. Its 1.3B model achieves an average re-compilability rate of 0.9227 and an average re-executability rate of 0.3432. When the model size is increased to 6.7B, the average re-executability rate further improves to 0.5378. Although larger models provide performance gains, they also significantly increase the computational resource requirements and training complexity.

DeepSeek-v3, with its large parameter scale and strong semantic understanding capabilities, outperforms other decompilation methods in re-compilability rates on both HumanEval and Exebench-test-real, achieving rates of 0.9649 and 0.8766, respectively. However, many of the decompiled results that can

be successfully compiled fail the I/O test. Only 39% and 41% of the compilable results pass the I/O tests. Further analysis of the failure cases reveals that while DeepSeek-v3 generally recovers the correct function semantics and structures, it often makes errors in restoring function argument lists or return values. This leads to situations where the decompiled results can be compiled but fail the I/O tests. This indicates that although large general models possess strong semantic capabilities, they still need fine-tuning for decompilation tasks to improve their practical usability.

Moreover, we observe that neural decompilation methods exhibit noticeably lower re-executability on samples compiled with optimization levels O1–O3 compared to O0. This trend is less apparent in Ghidra. Compiler optimizations often reorder or replace code logic, making it harder for neural models to learn stable mappings between assembly and source code under mixed optimization conditions. This highlights the need for future improvements, particularly under high optimization settings.

2) Comparison with Fine-Tuned LLM Methods: We further evaluate the applicability of small-scale large language models to the decompilation task. Specifically, we select LLaMA3.2-1B, Qwen2.5-0.5B, and Qwen2.5-Coder-1.5B as comparison models. All models are fine-tuned under the same experimental settings and on the same training dataset as DECodeT5, and are evaluated on identical evaluation benchmarks. The results are reported in Table I and Table II.

The experimental results show that small-parameter general-purpose LLMs (LLaMA3.2-1B and Qwen2.5-0.5B) exhibit limited performance on both benchmarks. On the ExeBench-test-real dataset, their executable-rate scores are only 0.2190 and 0.1873, respectively—substantially lower than that of DECodeT5. This gap primarily stems from the structural characteristics of pure decoder-only models. Such models specialize in autoregressive token generation within a single semantic space, but lack cross-attention mechanisms that explicitly model and align the source sequence. When the model size is small, it becomes even more difficult for them to implicitly learn accurate mappings across heterogeneous semantic spaces. In contrast, DECodeT5 adopts an encoder–decoder architecture, which explicitly models source–target alignment through separated encoding and decoding stages, and further incorporates a domain-specific assembly encoder that introduces strong structural inductive biases tailored for assembly-to-source mapping.

When model scale increases and additional programming-oriented fine-tuning is introduced, decoder-only models demonstrate a clear upward performance trend. For example, compared with Qwen2.5-0.5B, Qwen2.5-Coder-1.5B achieves approximately a 28% improvement in executable rate on HumanEval, and about a 98% improvement on ExeBench-test-real. A similar phenomenon is also observed in the LLM4Decompile model: increasing the model size from 1.3B to 6.7B yields over a 60% improvement in decompilation performance.

Overall, while decoder-only LLMs can indeed achieve better decompilation performance through substantial model scaling, such improvements typically come at the cost of significantly higher computational demands and larger training data requirements. These constraints make it challenging to efficiently deploy or fine-tune such models in resource-limited environments, and limit their ability to quickly adapt to different compilation configurations. The LLM4Decompile original paper [12] reports that when scaling to 33B parameters, even with sufficient hardware resources, communication overhead in distributed training becomes a major bottleneck limiting system efficiency. In contrast, DECodeT5 is designed as a lightweight, easily deployable, and highly transferable neural decompilation framework that directly addresses these practical limitations.

TABLE III
COMPARISON OF EDIT SIMILARITY OF DIFFERENT DECOMPILATION METHODS ON THE EXEBENCH-TEST-REAL BENCHMARK.

	Edit Similarity				
	O0	O1	O2	O3	Avg
Ghidra	0.3292	0.3229	0.3158	0.2987	0.3166
IDA pro	0.2266	0.2153	0.2157	0.1935	0.2127
SLaDe	0.5302	0.4636	0.4551	0.4068	0.4639
LLM4Decompile	0.6104	0.4974	0.4787	0.4155	0.5005
DeepSeek-v3	0.6794	0.5784	0.5512	0.5014	0.5776
DECodeT5	**0.6818**	**0.6008**	**0.5791**	**0.5072**	**0.5922**

B. Comparison of Code Readability Quality

To evaluate the quality of code generation, we measure the edit similarity between the decompiled results and the original functions on the Exebench-test-real dataset. To minimize the influence of code formatting (such as style and indentation) on the edit distance, we first apply clang-format to normalize all source code. We then flatten each function by removing indentation and non-semantic formatting before computing the edit similarity. The results under x86-64 architecture are shown in Table III.

The rule-based decompiler Ghidra achieves the lowest average edit similarity of 0.3166. This is because Ghidra relies heavily on subgraph matching within control flow graphs, which is effective only for standard structural patterns. For complex or unmatched structures, it often resorts to GOTO statements for control flow, leading to large structural and readability gaps between the generated code and the original functions.

Although IDA Pro achieves higher re-compilability and re-executability rates compared to Ghidra, its decompilation results contain a significant amount of tool-related noise, such as type casts and function calling conventions. As a result, its average edit similarity score is only 0.2127, which is lower than that of Ghidra.

In contrast, end-to-end neural decompilation methods significantly outperform traditional approaches in both structure recovery and readability. DECodeT5 achieves the highest average edit similarity of 0.5922, outperforming all baselines. Benefiting from CodeT5's strong code generation capabilities, DECodeT5 improves over SLaDe by approximately 27%, and surpasses LLM4Decompile-1.3B by 18%, respectively. DeepSeek-v3, leveraging its massive parameter count and training corpus, achieves an average similarity of 0.5776, second only to DECodeT5.

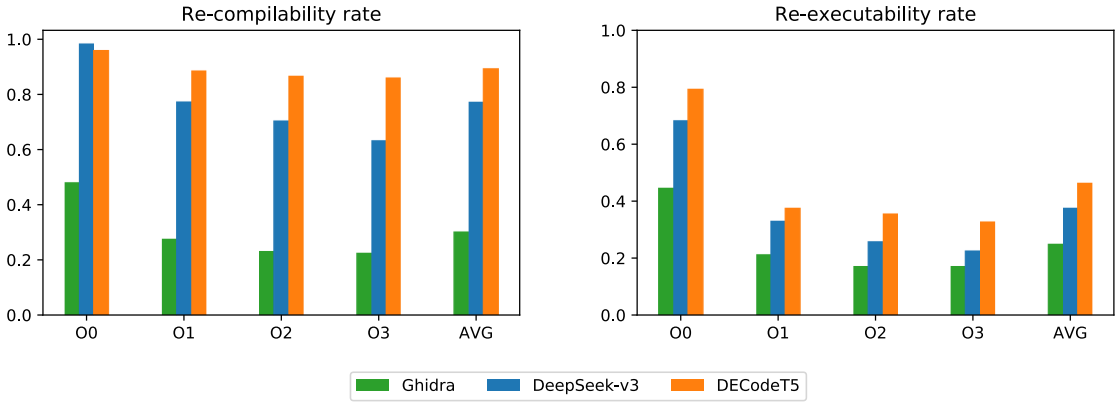


Fig. 6. Comparison of re-compatibility rates and re-executability rates of different decompilation methods under CLANG.

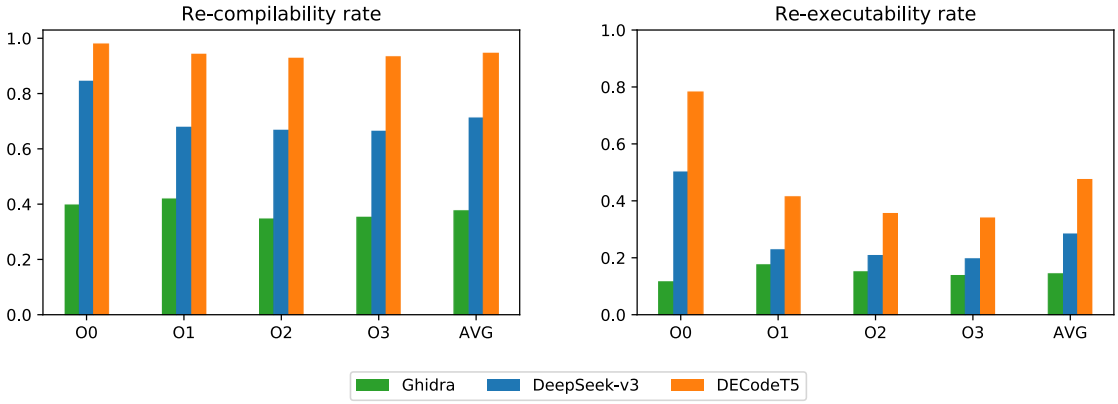


Fig. 7. Comparison of re-compatibility rates and re-executability rates of different decompilation methods under ARM.

These results demonstrate that DECodeT5 not only recovers execution semantics effectively but also generates code that is closely similar to the original source code, offering strong readability.

C. Comparison of Different Compiler and ISA

To evaluate the adaptability of DECodeT5 under different compilation conditions, we retrain the model using datasets compiled by different compilers and instruction set architectures (ISAs). Specifically, we compile the training data using clang-14.0.0 and arm-linux-gnueabihf-gcc-11.4.0, generating datasets targeting the CLANG compiler and ARM architecture, respectively.

For the CLANG compiler, we directly reuse the tokenizer and assembly encoder previously trained on the x86-64 architecture. For the ARM architecture, we build a new tokenizer and train a new assembly encoder tailored for ARM instructions. After training the model for decompilation, we evaluate its performance on the HumanEval dataset. The results are shown in Fig. 6 and Fig. 7.

The results demonstrate that DECodeT5 maintains strong performance under the new CLANG compiler. It achieved an average re-compilability rate of 0.8938 and an average re-executability rate of 0.4622 across three different optimization levels—both outperforming Ghidra and DeepSeek-v3. Under the high optimization level (O3), DECodeT5 still achieves a re-executability rate of 0.3276, reflecting strong generalization and stability. In contrast, Ghidra suffers from compiler-specific differences, with its performance dropping to 0.1707 under CLANG-O3 — a 24% decrease compared to its performance

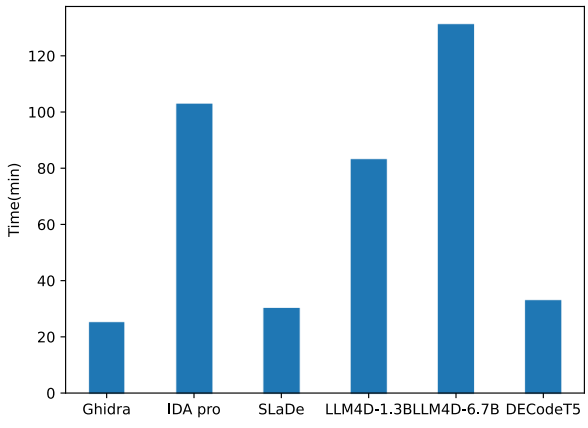


Fig. 8. Comparison of the decompilation time cost of different decompilation methods.

under GCC.

Under the ARM architecture, DECodeT5 also exhibits excellent performance, reaching an average re-compilability rate of 0.9459 and an average re-executability rate of 0.4722. By comparison, DeepSeek-v3, which lacks adaptation for ARM assembly code, shows a sharp performance decline to 0.2837, representing a 34% drop from its performance on x86-64.

In summary, these results confirm that DECodeT5 can be effectively adapted to different compilers and instruction set architectures. For a new compiler, the existing assembly encoder can be reused to achieve fast adaptation. For a new ISA, only the assembly encoder needs to be replaced and retrained, while the backend CodeT5 generator remains unchanged. This design significantly reduces the cost and complexity of adapting the model to diverse compilation conditions.

D. Comparison of Decompilation Time

To evaluate the decompilation efficiency of different methods, we conduct experiments on 1,744 functions from the Exebench-test-real dataset. All functions are compiled into binary files using the GCC compiler under the x86-64 architecture with the O0 optimization level. Each binary is then decompiled using different methods, and the total execution time is recorded, as shown in Fig. 8. Due to inconsistent runtime environments and high API latency, DeepSeek-v3 is excluded from the efficiency comparison to ensure fairness.

Among all methods, the traditional tool Ghidra is the fastest, completing decompilation in just 25 minutes, thanks to its mature and optimized rule-matching strategy. Our method, DECodeT5, with only 360M parameters, completes the task in 32 minutes, averaging about 1.11 seconds per function and achieving a throughput of 113 tokens per second.

In contrast, LLM4Decompile-1.3B takes 83 minutes, which is more than twice the time of DECodeT5. This is due to its larger model size (3.6× DECodeT5), increased attention heads, and larger hidden dimensions, which result in higher computational overhead. Additionally, its decoder-only architecture, built on DeepSeek-Coder, includes twice as many decoder layers, limiting parallelism and slowing inference. LLM4Decompile-6.7B, with even more parameters, required 131 minutes, four times longer than DECodeT5.

As an interactive reverse engineering tool, IDA Pro relies on launching the idat background process to execute decompilation scripts, which significantly limits its efficiency. In our experiments, its decompilation time costs 103 minutes, even exceeding that of LLM4Decompile-1.3B.

Overall, DECodeT5 delivers comparable speed to traditional tools like Ghidra, while maintaining strong decompilation quality. It is well-suited for deployment in resource-constrained environments.

E. Comparison of Different Input Lengths on Performance

To evaluate the impact of input length on decompilation performance, we conduct additional experiments across five token ranges (0–128, 128–256, 256–384, 384–512, and >512). Specifically, we use the x86 binaries compiled with gcc-O0 from ExeBench-test-real as the benchmark, partition samples according to the tokenized length produced by DECodeT5, and measure the re-executability of the decompiled code generated by both DECodeT5 and Ghidra. The results are shown in Figure 9.

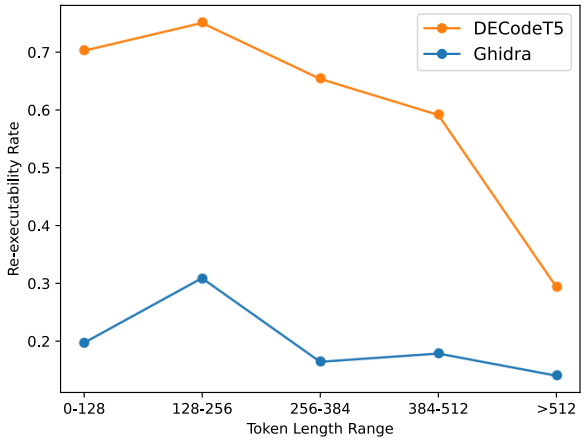


Fig. 9. Comparison of re-executability rates in different input length ranges.

Benefiting from the relative positional encoding used in the assembly encoder, DECodeT5 can process inputs exceeding 512 tokens during inference, even though the maximum training sequence length is capped at 512. Overall, DECodeT5 exhibits a gradual performance decline as sequence length increases: within 512 tokens, the re-executability remains above 0.6, but when the length exceeds 512, the metric drops sharply to around 0.3. This degradation mainly stems from the lack of long-sequence training data and the difficulty of modeling long-range dependencies.

In contrast, the rule-based traditional decompiler Ghidra shows more stable performance across different sequence lengths, with relatively small fluctuations. However, because it generates code that heavily depends on binary-specific custom types, its outputs often fail to compile successfully, resulting in a substantially lower overall re-executability compared with DECodeT5.

Although DECodeT5 experiences a performance drop on extremely long sequences, its performance within the 512-token range remains significantly stronger than that of traditional approaches. Combined with its small model size and robustness across compilation settings, DECodeT5 demonstrates strong potential as an alternative to conventional decompilation tools in diverse compilation scenarios.

F. Ablation

To evaluate the impact of the assembly encoder and code generation model on the decompilation task, we trained three additional models for comparison with DECodeT5:

- **Transformer:** Using the same architecture as CodeT5-base but starts from scratch with no pre-trained weights.
- **CodeT5:** Using the CodeT5-base model with pre-trained weights and is fine-tuned for decompilation.

TABLE IV
COMPARISON OF RE-EXECUTABILITY RATES OF DIFFERENT ABLATION METHODS ON THE HUMANEVAL BENCHMARK.

	Re-executability Rate				
	O0	O1	O2	O3	Avg
Transformer	0.5068	0.1917	0.1917	0.1700	0.2650
CodeT5	0.7320	0.3214	0.2720	0.2456	0.3927
DECodeT5_wo	0.7423	0.3871	0.3306	0.3019	0.4404
DECodeT5	**0.7835**	**0.4643**	**0.3813**	**0.3412**	**0.4923**

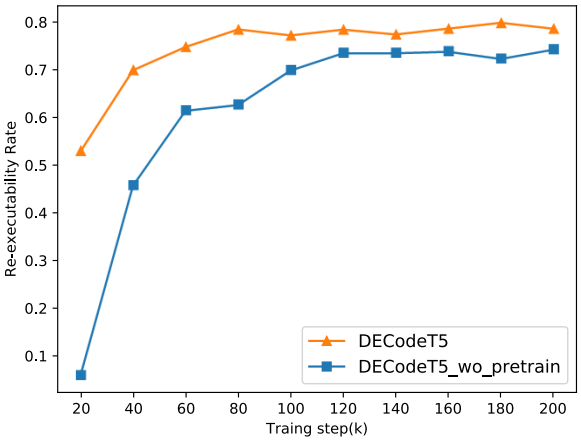


Fig. 10. Comparison of re-executability rates in different training steps.

- **DECodeT5_wo:** Using the full DECodeT5 architecture. The CodeT5-base weights are loaded, while the assembly encoder is randomly initialized without pre-training.

All models share the same hyperparameter settings during training. We evaluated them on the HumanEval dataset by measuring the re-executable rate of their decompiled outputs under various optimization levels. The results are shown in Table IV.

The results reveal that CodeT5, which benefits from pre-trained weights, outperforms the uninitialized Transformer, achieving a 48% higher average re-executable rate, highlighting the advantage of CodeT5's strong code generation capabilities.

However, CodeT5 lacks any understanding of assembly semantics. Its tokenizer tends to over-fragment tokens like register names, and since its encoder and decoder share the embedding layer, learning assembly semantics can disrupt the pre-trained embeddings—ultimately limiting performance.

DECodeT5_wo improves executable rate by 12% over CodeT5, thanks to its dedicated assembly encoder. However, it requires significantly more training time. As shown in Fig. 10, it takes 200,000 training steps to reach the performance DECodeT5 achieves in just 60,000 steps—a threefold increase in training time.

DECodeT5, by combining a code generation model with a pre-trained assembly encoder, achieves the best overall performance and fastest convergence. Its average executable rate is 85% higher than Transformer, 25% higher than CodeT5, and 11% higher than DECodeT5_wo.

These results show that integrating a pre-trained assembly encoder into a code generation model enables more effective and efficient learning of the mapping between assembly and source code for decompilation tasks.

VI. CASE STUDY

In this section, we evaluate the decompilation effectiveness of DECodeT5 using the case study introduced in Section II. The source code shown in Fig. 1(a) is compiled using the GCC compiler under the x86-64 architecture with both O0 (no optimization) and O2 (high optimization) settings. The resulting assembly code is then decompiled using DECodeT5, and the results are presented in Fig. 11.

The function `func0` in Fig. 1(a) is designed to check whether the angle brackets in a given string are correctly matched—returning 1 if they match and -1 otherwise. As shown, DECodeT5 accurately recovers the original execution semantics under both the O0 and O2 compilation settings. The decompiled code successfully passes all input/output validation. Although the O2 version slightly modifies the conditional logic and uses a ternary conditional operator to determine the return value, the overall control flow and logic remain faithful to the original function.

Compared to the decompilation outputs from Ghidra and ChatGPT-4o, DECodeT5 produces more concise and readable code that more closely resembles the structure of the original source. This highlights DECodeT5's superior ability to generate clean and human-friendly decompiled code, even under high optimization conditions.

VII. DISCUSSION

This work introduces DECodeT5, a neural decompilation framework designed to reconstruct function-level semantic structure from low-level assembly instructions. The framework focuses on learning a robust mapping between instruction sequences and high-level program semantics. Unlike general code translation models, DECodeT5 replaces the original CodeT5 embedding layer with a dedicated assembly encoder, enabling finer-grained modeling of assembly semantics. This design effectively narrows the semantic gap between assembly and source code, accelerates convergence during end-to-end training, and substantially improves the overall quality of decompilation. Experimental results demonstrate that DECodeT5 consistently outperforms traditional decompilation baselines and small fine-tuned language models under various compiler optimization settings.

Despite its strong performance, DECodeT5 still exhibits several limitations. First, the model is trained with a maximum sequence length of 512 tokens. Although the use of relative positional encoding allows the model to process longer inputs during inference, the lack of long-sequence training samples limits its ability to capture long-range dependencies. As a result, decompilation accuracy declines notably when the input length exceeds 512 tokens, making insufficient long-range semantic modeling one of the primary bottlenecks. Second, DECodeT5 is designed for intra-procedural decompilation and

```
int func0(char *s) {
    int count = 0;
    int i;
    for (i = 0; s[i]!= '\0'; i++) {
        if (s[i] == '<') count++;
        if (s[i] == '>') count--;
        if (count < 0) return -1;
    }
    if (count!= 0) return -1;
    return 1;
}
```

(a) DECodeT5 (GCC O0)

```
int func0(char *s) {
    int i, j;
    for (i = 0, j = 0; s[i]!= '\0'; i++) {
        if (s[i] == '<')
            j++;
        else if (s[i] == '>')
            j--;
        if (j < 0) return j;
    }
    return j == 0? 1 : -1;
}
```

(b) DECodeT5 (GCC O2)

Fig. 11. DECodeT5 decompilation results under O0 and O2.

does not model cross-function optimization behaviors. Compiler transformations such as function inlining, interprocedural constant propagation, cross-module dead-code elimination, and link-time optimization (LTO) reshape global control-flow and data-flow structures and blur natural function boundaries. These transformations break the semantic boundary assumptions on which function-level models rely, thereby reducing the stability and accuracy of decompilation.

For future work, we plan to extend the framework in two directions. The first is to enhance long-sequence modeling. We will conduct targeted fine-tuning using long assembly sequences to improve adaptability to real-world binaries with large functions. In addition, we will explore hierarchical encoding structures and incorporate Transformer-XL–style memory mechanisms to explicitly model long-range dependencies across segments. The second direction is to support real-world binaries that contain LTO and other advanced optimizations. We intend to integrate static analysis results—such as call-graph and data-flow information—as additional structured inputs to help the model identify and represent cross-procedural optimization behavior.

From a practical standpoint, DECodeT5 demonstrates strong feasibility and utility in real decompilation workflows. Its improved readability and semantic recovery make it suitable for integration as a plugin or semantic assistant in existing reverse-engineering toolchains, helping analysts efficiently recover structural information. Moreover, with a model size of only 360 MB and high inference efficiency, DECodeT5 can be deployed in automated analysis systems for large-scale binary processing without heavy hardware requirements, offering considerable flexibility for real-world deployment. In addition, its modular design enables rapid adaptation to new instruction set architectures or proprietary compilers, without the need to redesign rule sets as in rule-based decompilers, or to allocate substantial computational resources and training time as required by large LLM-based approaches.

VIII. Related Work

A. Tranditional Decompilation

Decompilation, a fundamental technique in reverse engineering, was first introduced by Cifuentes in 1994 [26]. Traditional decompilation methods primarily rely on rule-based matching. These approaches typically start by using platform-specific disassemblers to convert binary machine code into assembly instructions and control flow graphs (CFGs). They then infer high-level source code types through pattern matching and reconstruct control flow structures such as branches and loops using structural analysis techniques [27].

Among these steps, control flow reconstruction is particularly critical. Commonly, graph pattern matching is used to identify subgraphs in the CFG that correspond to control structures in high-level languages. Commercial reverse engineering tools like IDA Pro's Hex-Rays decompiler also rely on enhanced structural analysis to improve control flow reconstruction [28]. However, when the CFG contains structures that cannot be matched to known patterns, these tools often fall back on using GOTO statements. As program complexity increases, this fallback mechanism significantly degrades the readability and quality of the decompiled source code.

Research in this area mainly focuses on optimizing matching rules and code generation methods to improve accuracy and code quality [29]–[33]. A typical approach is to enhance readability by eliminating GOTO statements [11], [34], [35]. More recently, SAILR [10] incorporated deeper compiler insights into the matching process, refining GOTO elimination by aligning it with compiler-specific generation patterns.

B. Neural Decompilation

In recent years, an increasing number of studies have applied neural networks to learn the complex relationship between low-level assembly instructions and high-level programming languages [8], [17], [36]–[38]. One prominent approach views decompilation as a translation problem, employing sequence-to-sequence neural machine translation (NMT) models to learn semantic mappings from assembly to source code.

Deborah et al. [39] uses an RNN-based encoder-decoder model to map assembly code to LLVM intermediate representation, reducing the complexity of directly generating high-level source code. Coda [18] leverages Tree-LSTM [40] and an attention mechanism [41], [42] to learn assembly semantics and reconstruct the abstract syntax tree (AST) of the source code, enabling better recovery of complex expressions and syntax structures. SLaDe [9] utilizes BART for translating assembly to C code, and employs PsycheC to recover variables and type information in the generated code.

With the rise of large language models (LLMs), significant progress has been made in code understanding and generation [43]–[46]. These models leverage deep decoder stacks to generate more accurate and readable source code. LLM4Decompile [12], for example, achieves high recompile and re-executability rates by fine-tuning DeepSeek-coder on both compilable and executable datasets. ReF Decompile [13] further advances this by using LLMs for type inference and combining them with data segment analysis to recover missing constants and strings in assembly code.

In addition to direct decompilation, some works enhance traditional tools by recovering auxiliary information such as variable names, types, and function names for better readability [47]–[51]. Typilus [52] uses a variant of gated graph neural networks (GGNN) [53] to infer variable types in decompiled outputs. DIRTY [54] trains a Transformer model on a large corpus of GitHub code, enabling end-to-end recovery of variable names and types.

IX. Conclusion

This paper presents DECodeT5, a novel neural decompilation framework for the C programming language. DECodeT5 introduces a pre-trained assembly encoder that replaces the embedding layer of the original CodeT5 encoder, thereby incorporating assembly semantics into the code generation model. This innovative design significantly enhances the model's ability to learn semantic mappings between assembly code and source code in end-to-end decompilation tasks. Thanks to its modular architecture—separating the assembly encoder and code generator—DECodeT5 supports rapid adaptation across different compilers and instruction set architectures, greatly improving its generality and scalability.

Experimental results demonstrate that DECodeT5 achieves the best performance in terms of re-executability rate and edit similarity across various compiler optimization levels on benchmark datasets. It significantly outperforms existing methods such as Ghidra, DeepSeek-v3, and LLM4Decompile. With only 360 million parameters, DECodeT5 requires substantially fewer computational resources than mainstream large language models, enabling faster inference and efficient deployment, especially in resource-constrained environments.

Overall, DECodeT5 strikes a strong balance between decompilation performance, model compactness, and cross-platform adaptability, offering an efficient and practical solution for intelligent decompilation across diverse compilation scenarios.

Acknowledgments

This work is supported by the National Defense Basic Scientific Research Program of China (No. JCKY2023603C043), the Key RD Plan of Heilongjiang Province (No. 2022ZX01C01), and Natural Science Foundation of Heilongjiang Province of China (No. LH2024F023).

References

[1] Y. Feng, D. Teng, Y. Xu, H. Mu, X. Xu, L. Qin, Q. Zhu, and W. Che, "Self-constructed context decompilation with fined-grained alignment enhancement," in *EMNLP*, 2024, pp. 6603–6614.

[2] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, and X. Zhang, "Nova$^+$: Generative language models for binaries," *CoRR*, vol. abs/2311.13721, 2023.

[3] X. Xu, Z. Zhang, S. Feng, Y. Ye, Z. Su, N. Jiang, S. Cheng, L. Tan, and X. Zhang, "Lmpa: Improving decompilation by synergy of large language model and program analysis," *CoRR*, vol. abs/2306.02546, 2023.

[4] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *IEEE Symposium on Security and Privacy, SP*, 2016, pp. 158–177.

[5] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 95–110.

[6] N. S. Agency. (2024) Ghidra. [Online]. Available: https://ghidra-sre.org/

[7] H. rays. (2023) Ida pro. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[8] I. Hosseini and B. Dolan-Gavitt, "Beyond the C: retargetable decompilation using neural machine translation," in *NDSS Workshop on Binary Analysis Research*, 2022.

[9] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. P. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," in *CGO*, 2024, pp. 67–80.

[10] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O'Kain, D. Miao, T. Bao, A. Doupé, Y. Shoshitaishvili, and R. Wang, "Ahoy sailr! there is no need to DREAM of C: A compiler-aware structuring algorithm for binary decompilation," in *USENIX Security*, 2024.

[11] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations," in *NDSS*, 2015.

[12] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," in *EMNLP*, 2024, pp. 3473–3487.

[13] Y. Feng, B. Li, X. Shi, Q. Zhu, and W. Che, "Ref decompile: Relabeling and function call enhanced decompile," *CoRR*, vol. abs/2502.12221, 2025.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008.

[15] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *EMNLP*, 2021, pp. 8696–8708.

[16] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. de Souza Magalhães, and M. F. P. O'Boyle, "Exebench: an ml-scale dataset of executable C functions," in *MAPS*, 2022, pp. 50–59.

[17] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *CoRR*, vol. abs/1905.08325, 2019.

[18] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "Coda: An end-to-end neural program decompiler," in *NeurIPS*, 2019, pp. 3703–3714.

[19] C. Wang, K. Cho, and J. Gu, "Neural machine translation with byte-level subwords," in *AAAI*, 2020, pp. 9154–9160.

[20] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, J. Burstein, C. Doran, and T. Solorio, Eds., 2019, pp. 4171–4186.

[21] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *CoRR*, vol. abs/1910.03771, 2019.

[22] A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimarães, and F. M. Q. Pereira, "ANGHABENCH: A suite with one million compilable C benchmarks for code-size reduction," in *CGO*, 2021, pp. 378–390.

[23] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *ACL*, 2020, pp. 7871–7880.

[24] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder:

When the large language model meets programming - the rise of code intelligence," *CoRR*, vol. abs/2401.14196, 2024.

[25] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Wang, J. Chen, J. Chen, J. Yuan, J. Qiu, J. Li, J. Song, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Wang, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Wang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Zhang, R. Pan, R. Wang, R. Xu, R. Zhang, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Pan, T. Wang, T. Yun, T. Pei, T. Sun, W. L. Xiao, and W. Zeng, "Deepseek-v3 technical report," *CoRR*, vol. abs/2412.19437, 2024.

[26] C. Cifuentes, "Reverse compilation techniques," 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:110021381

[27] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Softw. Pract. Exp.*, vol. 25, no. 7, pp. 811–829, 1995.

[28] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *USENIX Security*, 2013, pp. 353–368.

[29] M. V. Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *11th Working Conference on Reverse Engineering, WCRE 2004, Delft, The Netherlands, November 8-12, 2004*. IEEE Computer Society, 2004, pp. 27–36.

[30] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt, "Learning to analyze binary computer code," in *AAAI*, D. Fox and C. P. Gomes, Eds., 2008, pp. 798–804.

[31] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *CAV*, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806, 2011, pp. 463–469.

[32] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: learning to recognize functions in binary code," in *USENIX Security*, K. Fu and J. Jung, Eds., 2014, pp. 845–860.

[33] Z. Tan, Y. Chon, M. Kruse, J. Doerfert, Z. Xu, B. Homerding, S. Campanoni, and D. I. August, "SPLENDID: supporting parallel LLVM-IR enhanced natural decompilation for interactive development," in *ASPLOS*, 2023, pp. 679–693.

[34] Z. Liu and S. Wang, "How far we have come: testing decompilation correctness of C decompilers," in *ISSTA*, 2020, pp. 475–487.

[35] A. D. Federico, P. Fezzardi, and G. Agosta, "rev.ng: A multi-architecture framework for reverse engineering and vulnerability discovery," in *ICCST*, 2018, pp. 1–5.

[36] R. Liang, Y. Cao, P. Hu, J. He, and K. Chen, "Semantics-recovering decompilation through neural machine translation," *CoRR*, vol. abs/2112.15491, 2021.

[37] Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in *ACSAC*, 2022, pp. 508–518.

[38] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled C code with large language models," *CoRR*, vol. abs/2310.06530, 2023.

[39] D. S. Katz, J. Ruchti, and E. M. Schulte, "Using recurrent neural networks for decompilation," in *SANER*, 2018, pp. 346–356.

[40] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *ACL*, 2015, pp. 1556–1566.

[41] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *ICLR*, 2018.

[42] L. Dong and M. Lapata, "Language to logical form with neural attention," in *ACL*, 2016.

[43] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. M. V, J. T. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" *Trans. Mach. Learn. Res.*, vol. 2023, 2023.

[44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," *CoRR*, vol. abs/2308.12950, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[45] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with LLM," in *NDSS*, 2024.

[46] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, T. Hennigan, E. Noland, K. Millican, G. van den Driessche, B. Damoc, A. Guy, S. Osindero, K. Simonyan, E. Elsen, J. W. Rae, O. Vinyals, and L. Sifre, "Training compute-optimal large language models," *CoRR*, vol. abs/2203.15556, 2022.

[47] A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: a machine translation approach," in *ICPC*, 2018, pp. 20–30.

[48] F. Artuso, G. A. D. Luna, L. Massarelli, and L. Querzoni, "Function naming in stripped binaries using neural networks," *CoRR*, vol. abs/1912.07946, 2019.

[49] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming," in *ASE*, 2019, pp. 628–639.

[50] Z. Zhang, Y. Ye, W. You, G. Tao, W. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "OSPREY: recovery of variable and data structure via probabilistic analysis for stripped binary," in *IEEE Symposium on Security and Privacy, SP*, 2021, pp. 813–832.

[51] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *ICSE*, 2022, pp. 2327–2339.

[52] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," in *PLDI*, 2020, pp. 91–105.

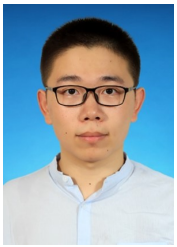
[53] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *ICLR*, Y. Bengio and Y. LeCun, Eds., 2016.

[54] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *USENIX Security*, 2022, pp. 4327–4343.

Li Liu received the B.S. degree in software engineering from Weihai Campus of Harbin Institute of Technology, Weihai, China, in 2019, the M.S. degree in computer technology from Harbin Institute of Technology, Harbin, China, in 2021. He is currently pursuing the Ph.D. degree in cyberspace security with Harbin Institute of Technology, Harbin, China. His current research interests include the binary code similarity and firmware vulnerability mining.

Fanghui Sun received the Ph.D. degree in Cyberspace Security from Harbin Institute of Technology, Harbin, China, in 2021. She is currently an Associate Researcher of Harbin Institute of Technology. Her research interests include cyberspace security countermeasure, artificial intelligence security, reverse analysis of network protocols, and reverse engineering.

Shen Wang received the B.S. and M.E. degrees in electrical engineering and information technology from TU Dresden University of Technology, Dresden, Germany, in 2003 and 2007, respectively, and Ph.D. degree in computer science and technology from Harbin Institute of Technology, Harbin, China, in 2012. Currently, he is a professor in the School of Cyberspace Science, Harbin Institute of Technology. His research interests include adversarial attack and defense based on machine learning, image disguise and digital forensics.



Xunzhi Jiang received the B.S. degree in software engineering from Harbin Engineering University, Harbin, China, in 2018. He is currently pursuing the Ph.D. degree in cyberspace security with Harbin Institute of Technology, Harbin, China. His current research interests include deep learning, binary code similarity and firmware vulnerability mining.