Boosting Neural Networks to Decompile Optimized Binaries

Ying Cao, Ruigang Liang*
{caoying,liangruigang}@iie.ac.cn
SKLOIS, IIE, CAS‡
School of CyberSecurity, UCAS§
Beijing, China

Kai Chen†
chenkai@iie.ac.cn
SKLOIS, IIE, CAS‡
School of CyberSecurity, UCAS§
BAAI¶
Beijing, China

Peiwei Hu
hupeiwei@iie.ac.cn
SKLOIS, IIE, CAS‡
School of CyberSecurity, UCAS§
Beijing, China

ABSTRACT

Decompilation aims to transform a low-level program language (LPL) (eg., binary file) into its functionally-equivalent high-level program language (HPL) (e.g., C/C++). It is a core technology in software security, especially in vulnerability discovery and malware analysis. In recent years, with the successful application of neural machine translation (NMT) models in natural language processing (NLP), researchers have tried to build neural decompilers by borrowing the idea of NMT. They formulate the decompilation process as a translation problem between LPL and HPL, aiming to reduce the human cost required to develop decompilation tools and improve their generalizability. However, state-of-the-art learning-based decompilers do not cope well with compiler-optimized binaries. Since real-world binaries are mostly compiler-optimized, decompilers that do not consider optimized binaries have limited practical significance. In this paper, we propose a novel learning-based approach named *NeurDP*, that targets compiler-optimized binaries. *NeurDP* uses a graph neural network (GNN) model to convert LPL to an intermediate representation (IR), which bridges the gap between source code and optimized binary. We also design an Optimized Translation Unit (OTU) to split functions into smaller code fragments for better translation performance. Evaluation results on datasets containing various types of statements show that *NeurDP* can decompile optimized binaries with 45.21% higher accuracy than state-of-the-art neural decompilation frameworks.

ACM Reference Format:
Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. 2022. Boosting Neural Networks to Decompile Optimized Binaries. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3564625.3567998

1 INTRODUCTION

In recent years, deep learning has made remarkable achievements in the fields of code comprehension and reverse engineering. Some work is devoted to using neural networks to learn representations of source code, such as GitHub Copilot [3] and CodeBERT [14], which are widely used in automatic code generation/completion and automatic comment generation. Similarly, many previous studies leverage neural networks to learn binary or assembly code representation. They perform well on binary-based downstream analysis tasks, including code clone detection [11], malicious code detection [12], and disassembly [28]. The application of deep learning in software analysis and software reverse engineering significantly reduces human resources and time costs, no matter from the view of developers or analysts. In addition, compared to traditional tools, the faster speed of deep neural-based disassembly approaches [28] makes them a powerful engine for downstream models like malware classification. It is meaningful to study how to make neural network (NN) models work well in software reverse engineering and software analysis.

Decompilation is a core technology in software reverse engineering and software analysis (e.g., vulnerability discovery [13, 30] and malware analysis [17, 34]), especially adopted in the analysis of commercial software whose source code is not available. The decompilation process can be defined as translating a low-level PL (LPL) (e.g., binary file) into its functionally-equivalent high-level PL (HPL) (e.g., C/C++). Developing a traditional decompiler requires much work to manually reverse multiple binaries to analyze and summarize the heuristic rules used in the decompiler. The well-known open-source decompiler RetDec [21] has hundreds of developers who contributed code to it. However, since they released the prototype in 2017, the decompilation performance is still unsatisfactory [25]. Due to the limited number of binaries analyzed by PL experts, the decompilation rules are often incomplete. Moreover, the rules might change with instruction set architecture, compilers, and HPL. It is hard to construct a complete decompilation rule set. Therefore, decompilers built on human-defined rules need to iterate continuously by summarizing the rules and collecting feedback on the errors encountered by users. With the help of neural-based approaches, expert efforts to generalize and revise rules could be largely reduced.

Several neural-based approaches [16, 19, 20] are explored to decompile LPL to HPL, hoping that the learning algorithms can automatically learn the mapping rules between LPL and HPL. They all used assembly code (ASM) as LPL and source code or abstract syntax tree (AST) as HPL to train an end-to-end model. These schemes perform various preprocessing on the input and output and design various model architectures according to code characteristics. However, they still suffer a severe drawback: *none of these works can appropriately handle the decompilation of optimized code.* As compiler optimization is ubiquitously used, the ability to

*Both authors contributed equally to this research.
†Corresponding Author.
‡Institute of Information Engineering, Chinese Academy of Sciences.
§University of Chinese Academy of Sciences
¶Beijing Academy of Artificial Intelligence

ACSAC '22, December 5–9, 2022, Austin, TX, USA

ACM ISBN 978-1-4503-9759-9/22/12.
https://doi.org/10.1145/3564625.3567998

decompile optimized code is essential for the practical application of neural-based approaches.

Through extensive analysis and experimentation, we found that it is not easy to train an end-to-end decompilation model that can handle optimized LPL. Since deep learning models are data-driven, a high-quality training set is critical to the model's performance. Most models used in the source code or reverse engineering domains rely on large, high-quality supervised or unsupervised datasets. In contrast, there are very few mature datasets in the field of decompilation, and datasets used in previous neural-based decompilation studies are not built for the decompilation of optimized LPL. Without a well-labeled dataset, it is difficult for the model to learn the mapping rules between HPL and LPL. Although many open-source projects exist in the real world, the source code and binary code cannot be completely matched at the statement level due to code optimization (e.g., dead code elimination), making it inaccurate to directly use source code as the labels for the optimized binaries. We summarize the challenges as follows.

Challenges. C1: It is well known that statements of HPL are often significantly refactored during compiler optimization, which makes it challenging to make an exact match between the semantics of LPL and HPL. For example, dead code elimination causes certain statements in HPL not to appear in LPL. Loop unwinding can cause some code to appear multiple times in the binary and only once in the source code. These optimization strategies all lead to the code structure and semantic information in the text level of LPL being quite different from HPL. In some cases, textually similar HPL codes (only some variable names differ) can correspond to completely different LPL codes, and vice versa. Therefore, it is not feasible to directly train end-to-end models using HPL as the label of LPL, which makes it challenging to capture the decompilation rules.

C2: Splitting LPL and HPL into code fragments with correct correspondence is a nontrivial task. Previous work typically utilizes functions or basic blocks (BBs) as input units for training neural models. However, the number of instructions in a function or a BB can be infinite (up to 1,000 instructions), which is hard to handle appropriately by neural network models. Therefore, it is essential to split the BB into finer-grained units, which can effectively reduce the model's difficulty in learning the decompiled rules. One straightforward method is to split the LPL or HPL based on debug information. However, the LPL and HPL mapped by the debug information are inaccurate, especially for optimized binaries. For example, the dead code in the HPL will also be mapped onto the LPL along with the live code. Another straightforward way is to set a maximum fragment length and split the basic block into fragments. However, several statements in a fragment may have over one independent feature. For example, there are three independent features (data flows) in the code segment "$a = 0; b = c + d; call(c);$". Thus it is difficult for the model to properly encode it into a single vector representing its function or semantics. Splitting data dependency graphs (DDG) may solve this problem, but it is also a difficult task. Worse still, the DDGs of LPL and HPL are quite different because of compiler optimization.

Our approach. In this paper, we propose an NN-based decompilation framework called NeurDP[1]. *NeurDP* uses a neural network model to translate LPL into an optimized IR (IR decompiler) to address C1 instead of directly translating LPL into HPL. As we know, the compiler first generates the intermediate representation (IR) code during the compilation process and performs most of the optimization strategies on the IR. Therefore, the structural differences between the optimized IR and LPL are much more minor than those between HPL and LPL. Compared to previous end-to-end neural decompilers, *NeurDP* can cope with the decompilation problem of compiler-optimized LPL. Finally, *NeurDP* converts IR statement to HPL statement directly.

Specifically, to train a well-performing IR decompiler model, we design a splitting technique called Optimal Translation Unit (OTU) to address C2. *OTU* splits BBs into smaller pairs of LPL and *HIR* fragments. The statements in each fragment have data dependencies and can be synthesized into one feature. *OTU* helps build a high-quality training set for our NN model.

To evaluate the accuracy of *NeurDP*, we use several programs randomly generated using our tool which is developed by cfile [2] and regular expressions, including 500 lines of code. Since our goal is to boost the neural network's ability for decompilation, we compare *NeurDP* with related studies using neural networks (e.g., Coda [16] and Neutron [24]). Experimental results show that *NeurDP* is 5.8%-27.8% more accurate than Coda on unoptimized code. Moreover, *NeurDP* can handle compiler-optimized code well, while Coda is incapable of action. *NeurDP* can decompile optimized binaries with 45.21% higher accuracy than another neural decompilation Neutron [24]. According to our evaluation, the introduction of *OTU* and IR mechanisms in our model improves the accuracy by 4.1%-71.23% compared to using the model directly in the optimized code.

Contributions. Our main contributions are outlined below:

- We design a novel neural machine decompilation technique. It is the first neural-based decompiler that can handle compiler-optimized code.
- We design an optimal translation unit (OTU) scheme, which can help other researchers form a sound dataset for training the IR decompilation model or other applications.
- We implement our techniques and conduct extensive evaluations. The results show that *NeurDP* is much better than the state-of-the-art neural-based decompilers, especially for optimized code. We release our dataset and the NN parameters on GitHub[2].

2 BACKGROUND

2.1 Compilation and Optimization

Compilation translates HPL (e.g., C/C++) into LPL (e.g., machine code) that can be run on the target CPU (e.g., X86, ARM). Due to the differences between the two PLs, unnecessary information (e.g., symbols) for the CPU is usually removed. Also, in this process, optimization technologies are designed to minimize or maximize some attributes of the executable program, e.g., to reduce the program's memory usage. Note that the execution results of the optimized target program should be the same as the original program without optimization. Optimization usually has several levels (e.g., from

[1]NeurDP (Neural Decompilation)
[2]https://github.com/zijiancogito/neur-dp-data.git

```c
#include <stdio.h>
#include <stdlib.h>
int f_scanf_nop(void)
{
   int var0;
   scanf("%d", &var0);
   return var0;
}

int f_rand(void)
{
   int var0 = rand();
   return var0;
}

int func1(int p0)
{
   int var0 = f_scanf_nop();
   int var1 = f_rand();
   int var2 = -123;
   var1 = var0 + p0 * var1;
   var1 = var1 / var0;
   var1 = var1 / -123;
   return var1;
}
```

(a) HPL

```
func1(int):
…
bl      f_scanf_nop()
str     w0, [sp, #8]
bl      f_rand()
str     w0, [sp, #4]
mov     w8, #-123
str     w8, [sp]
ldr     w9, [sp, #8]
ldur    w10, [x29, #-4]
ldr     w11, [sp, #4]
mul     w10, w10, w11
add     w9, w9, w10
str     w9, [sp, #4]
ldr     w9, [sp, #4]
ldr     w10, [sp, #8]
sdiv    w9, w9, w10
str     w9, [sp, #4]
ldr     w9, [sp, #4]
sdiv    w8, w9, w8
str     w8, [sp, #4]
ldr     w0, [sp, #4]
ldp     x29, x30, [sp, #16]
add     sp, sp, #32
ret
```

(b) LPL on O0

```
func1(int):
…
bl      f_scanf_nop()
mov     w20, w0
bl      f_rand()
madd    w8, w0, w19, w20
mov     w9, #65003
movk    w9, #57010, lsl #16
sdiv    w8, w8, w20
ldp     x20, x19, [sp, #16]
smull   x8, w8, w9
lsr     x9, x8, #63
asr     x8, x8, #36
add     w0, w8, w9
ldp     x29, x30, [sp], #32
ret
```

(c) LPL on O1

```
func1(int):
…
mov     w19, w0
adrp    x0, .L.str
add     x0, x0, :lo12:.L.str
sub     x1, x29, #4
bl      scanf
ldur    w20, [x29, #-4]
bl      rand
madd    w8, w0, w19, w20
mov     w9, #65003
movk    w9, #57010, lsl #16
sdiv    w8, w8, w20
ldp     x20, x19, [sp, #32]
ldp     x29, x30, [sp, #16]
smull   x8, w8, w9
lsr     x9, x8, #63
asr     x8, x8, #36
add     w0, w8, w9
add     sp, sp, #48
ret
```

(d) LPL on O2

Figure 1: An example showing different compiler optimization levels

`O0` to `O3` for the compiler gcc[3]). The higher the optimization level, the greater the difference between the compiled binary code and the source code. Optimization makes it more difficult to generate decompiled code using deep learning models. For example, the optimization could look for redundant operations among lines of code and combine them, or calculate some operations in the compiling time rather than the running time. This would change the structure of HPL. The optimization process is usually irreversible. Also, the optimization strategies are very diverse, even for the same type of operation. For example, in Figure 1, the target program languages after optimization for the division operations can have different forms when they have different types of operands. Specifically, if the operand is variable, the translated code is `sdiv`. Moreover, when the operand is changed to immediate, the target code is `mov, movk, smull, lsr, asr, add`, which does not even contain the division operation.

2.2 Decompilation

Decompilation is a technique that transforms a compiled executable program or ASM (LPL) into a functionally equivalent HPL [31]. As mentioned previously, to decompile code, analysts would make many heuristic rules to help lift binary code to source code [4, 6, 7, 21]. However, generalizing the rules is challenging since complex instruction set architectures (ISA), code structure, and optimization strategies. Experts need to summarize the code changes brought by many optimization strategies and handcraft the corresponding decompilation rules. For example, in Figure 1, using different optimization levels, the operation `var1=var1/-123;` could be compiled into different types of instructions, e.g., `mov, movk, smull, lsr, asr, add`. In this case, the developer needs to handcraft the rules to analyze the data dependencies of these instructions, and determine whether these instructions represent a division statement. The situation worsens when new operations are added, or new optimization strategies are developed, introducing new rules and impacting the old ones. What is more, this may mislead a neural-based decompiler to translate the similar code `smull, lsr, asr, add`, which does not mean division operation to `sdiv`. Obtaining a good model requires training on a large-scale dataset with high-quality labels. However, directly using the source code as the label for optimized binary decompilation is inaccurate due to the gap between the source code and the optimized code.

Existing decompilation tools typically design an intermediate representation (IR) as a bridge between the LPL and the HPL. While converting IR to HPL is a relatively easy task [5], the rules for translating LPL to IR rely on expert analysis and definitions. Moreover, as each tool proposes its own IR and has different definitions of micro-operations, rule-based decompilers suffer from poor generalizability and scalability. To solve these issues, researchers proposed neural-based decompilation [16, 19, 20, 24]. Katz et al. [19] adopted methods from NMT and formulated decompilation as a language translation task, aiming to overcome the bottleneck of rule-based approaches. Coda [20] and Neutron [24] are designed to learn the mapping rules automatically. The neural-based approaches bring a new idea to program decompilation. However, previous neural-based methods all learn a direct mapping from LPL to HPL or the abstract syntax trees (ASTs) of HPL. In addition, none of the current neural-based approaches can handle optimized code, mainly due to the unavailability of a high-quality dataset of optimized code.

3 APPROACH

We propose a novel neural decompilation approach *NeurDP* that can handle compiler-optimized code. *NeurDP* first translates LPL to *HIR* using GNN based IR decompiler model, and then recovers *HIR* code to HPL. Below we elaborate on the design of *NeurDP*.

3.1 Overview

The overview of *NeurDP* is shown in Figure 2, which aims to decompile the LPL into functionality-equal C-like HPL. And the detail of *NeurDP* is shown in Figure 6. Considering the large gap between the LPL and HPL, we introduce an IR named *HIR* as a bridge. IR

[3]https://gcc.gnu.org/

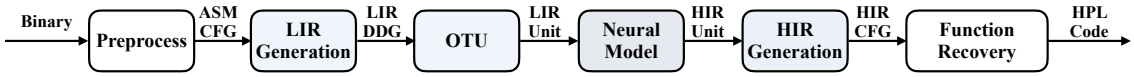


Figure 2: Overview of NeurDP

```
func4:
   274:   mov w21, w0
   …
   290:   bl #-592 <f_rand>
   294:   mov w25, w0
   298:   bl #-644 <f_scanf_nop>
   29c:   sub w8, w19, w23
   2a0:   mov w9, #871
   2a4:   mul w10, w24, w21
   2a8:   madd w8, w8, w9, w25
   2ac:   mul w19, w25, w10
   2b0:   mul w0, w19, w8
   2b4:   bl #-692 <f_printf>
   2b8:   mul w8, w19, w21
   2bc:   mul w8, w8, w20
   2c0:   mul w0, w8, w22
   …
   2d8:   ret
```

(a) LPL Code

```
func4
 …
 bl     x0_4, <f_rand>
 bl     x0_5, <f_scanf_nop>
 sub    x8_1, x2_0, x0_2
 mul    x10_1, x0_3, x0_0
 madd   x8_2, x8_1, 871, x0_4
 mul    x19_2, x0_4, x10_1

 mul    x0_6, x19_2, x8_2
 bl     x0_7, <f_printf>,x0_6
 mul    x8_3, x19_2, x0_0
 mul    x8_4, x8_3, x1_0
 mul    x0_8, x8_4, x0_1
 ret    x0_8
```

(b) LIR Code

```
@func4(%p0,%p1,%p2)
{
 …
 %call3 = call @f_rand()
 %call4 = call @f_scanf_nop()
 %sub = sub %p2, %call1
 %mul = mul %sub, 871
 %mul5 = mul %call2, %p0
 %add = add %mul, %call3
 %mul9 = mul %call3, %mul5
 %mul10 = mul %mul9, %add
 void = call @f_printf(%mul10)
 …

 %mul14 = mul %mul8, %call
 ret %mul14
}
```

(c) HIR Code

```
int func4(int p0, int p1, int p2)
{
 …
  int var3 = f_rand();
  int var4 = f_scanf_nop();
  int var5 = 871;
  int var6 = 88;
  p2 = (p2 - var1) * var5;
  var1 = p0 * var2;
  var5 = ((var1 * p0) * var3) * p1;
  p2 = ((p2 + var3) * var1) * var3;
  f_printf(p2);
  var3 = ((var3 + var6) - p2) - var5;

  var3 = var0 * var5;
  return var3;
}
```

(d) HPL Code

Figure 3: Example of relationship between HPL, HIR, LIR, and LPL

Table 1: Part of the rules from HIR to HPL

HIR	HPL
%result = sub %1, %2	result = v1 - v2;
%result = add %1, %2	result = v1 + v2;
%result = call f_printf, %1, %2	result = f_printf(v1, v2);
void = call f_printf, %1, %2	f_printf(v1, v2);
ret %1	return v1;

is optimized by the compiler front end. Using the optimized IR as the model's target can reduce the difficulty of model learning since the model no longer needs to learn to reverse the optimization strategies in the compiler's front end.

In the data construction phase, we first disassemble the binary file, identify the code sections from the binary and retrieve the assembly code of all functions. Then, *NeurDP* gets the control flow graph (CFG) for each function and the assembly code of each basic block following previous work [29]. Next, *NeurDP* performs static single assignment (SSA) and data dependency analysis on the LPL in each basic block. We do not use LPL directly as the model's input. Instead, we prefer to use the representation of the SSA form, which is a low-level intermediate representation (*LIR*) (see Section 3.2). After generating *LIR*, we further analyze the data dependency between *LIR* code to obtain the data dependency graph (DDG) of *LIR* within the basic block.

In the model processing phase, we design a neural model to translate *LIR* into *HIR* (Section 3.3), including the following two steps. ***Step 1: Model Training.*** Firstly, *NeurDP* prepares a suitable dataset for model training, which contains pairs of *LIR* and *HIR*. *NeurDP* splits the basic block into smaller snippets using Optimal Translation Unit (OTU). The *LIR* and *HIR* pair in the corresponding units are functionally equivalent, which makes it easier for the model to learn the transform rules. Secondly, we design and train a neural model based on graph neural network [23] model to generate *HIR* code. We describe the detailed design of *LIR*, *HIR* and *OTU* in Section 3.2 and the construction of training datasets and the model architecture in Section 3.3. ***Step 2: Model Translation.*** This step includes the recovery and reorganization of basic blocks in CFG. (i) Recovery: *NeurDP* recovers statements in basic blocks in this step. Firstly, *NeurDP* uses *OTU* to divide the basic block into units and get the DDG of each unit. Then, *NeurDP* uses the trained model to translate *LIR* DDG units to *HIR* code templates. After that, *NeurDP* fills the analyzed local variables from *LIR* into the *HIR* templates based on data flow analysis. We detail the method of operands recovery in Section 3.4. (ii) Reorgnization: In this step, we sort these *HIR* snippets based on the position of its corresponding *LIR* in the basic block to recover the complete *HIR* basic block. At last, we recover the CFG of *HIR* (HIR-CFG) based on the CFG of LPL.

In the HPL generation phase, *NeurDP* lifts the *HIR* to HPL. A complete function includes control structures, statements, and function signatures. Translating statements from *HIR* to HPL is not a difficult task, so we make some rules. Table 1 shows some of the rules. For the recovery of control flow and function signatures, many other researchers are focusing on these problems, and we use existing studies [9, 35] for these two parts. With the function signatures, the statements within each basic block, and the control structures between the basic blocks, we can construct a complete HPL function.

3.2 Dataset Construction

As mentioned previously, we cannot directly use a function's HPL and LPL as input-output pairs for the model. The main obstacles lie in that (i) the instructions in HPL and LPL have poor correspondence (e.g., redundant or missing operations), and (ii) each function has many instructions, making it difficult for the neural network models to learn. In order to solve the problems and facilitate effective learning, we introduce an intermediate representation (*HIR*) that has better instruction correspondence with the LPL and use *OTU* to split the basic blocks into smaller units.

Table 2: Some instruction templates of LIR

	LPL	LIR
Return	`ret`	`ret x0`
Unconditional Branch	`bl label`	`bl x0, label[, SRC [, SRC...]]`
Conditional Branch	`cmp SRC, SRC` `b.COND label1`	`b COND, label1, label2, SRC, SRC`
Store Register	`str SRC, DST`	`str DST, SRC`
Arithmetic (shifted register)	`add DST, SRC, SRC, [lsl] IMM`	`add DST, SRC, SRC` `lsl DST, IMM`
Move(wide immediate)	`add DST, SRC, SRC` `mov DST, IMM1`	`movk DST, IMM2, [lsl] 16|32) +IMM1`
WZR|XZR Register	`mov DST, WZR|XZR IMM`	`mov DST, 0`
Conditional Comparison	`ccmn SRC, SRC, IMM, cond`	`ccmn nzcv, cond, SRC, SRC, IMM`

label: jump address. **label1**: address of the next instruction. **IMM**: immediate. **nzcv**: condition flags.

Table 3: HIR Syntax Templates

LLVM IR	NeurDP HIR
<result> = mul <ty> <op1>, <op2>	<result> = mul <op1>, <op2>
<result> = add nuw nsw <ty> <op1>, <op2>	<result> = add <op1>, <op2>
<result> = fsub [fast-math flags]* <ty> <op1>, <op2>	<result> = fsub <op1>, <op2>
<result> = icmp <cond> <ty> <op1>, <op2>	<result> = icmp <cond> <op1>, <op2>
switch <intty> <value>, label <defaultdest> [<intty> <val>, label <dest> ...]	switch <value>, <defaultdest> [<val>, <dest> ...]

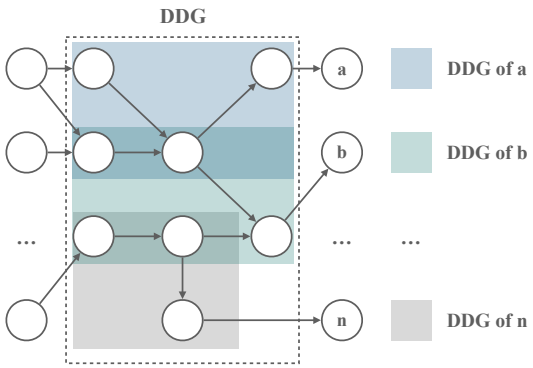


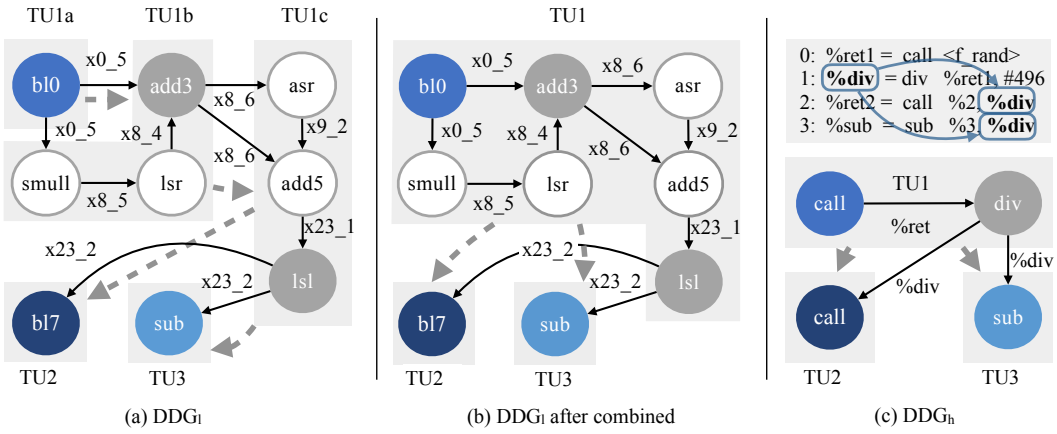
Figure 4: Black box of basic block

Intermediate Representation. *LIR* is lifted from LPL by removing machine-related features from LPL, such as registers, designed as the model's input. To get *LIR*, we first change LPL to SSA form, then use optimization strategies similar to constant (register) propagation to eliminate as many registers as possible. *LIR* maintains almost the same syntax as LPL ($< opcode > < op_{des} >, < op_{src1} >, ...$). Table 2 shows some of the syntax templates of *LIR*. For *HIR*, we extract the operands and opcodes from LLVM IR and rewrite them automatically according to the syntax ($< op_{des} >=< opcode > < op_{src1} >, ...$), which is a simplified scheme of LLVM IR. It is feasible to directly use LLVM IR instead of HPL as the model's output. However, a model that converts *LIR* to LLVM IR instead of *HIR* needs to learn too much additional information (like data types), which could complicate the model structure and make training such a model extremely difficult. Therefore, we choose to construct the model that converts *LIR* to *HIR*, which is relatively simple (though training this model is still not straightforward). Table 3 lists parts of the instruction templates we use for *HIR* and corresponding LLVM IR. Figure 3 (c) shows the *HIR* generated by *NeurDP*.

Optimal Translation Unit. *NeurDP* splits the basic block into smaller units that could let the model learn the mapping rules between *LIR* and *HIR* instructions easily. An unit in *LIR* should be functionally equivalent to the corresponding unit in *HIR*. One may use a fixed-length translation unit (TU) to spill a function into units. However, the units generated in this way may not be functionally equivalent. For example, in Figure 3, the `madd` instruction in (b) corresponds to the computation of `%mul` and `%add` in (c). Nevertheless, the two instructions are located far away and hard to include in a fixed-length TU. Also, a large size of the TU would include unrelated instructions that cannot be paired.

We assume that a basic block is a black box, as shown in Figure 4. We find that most optimization strategies do not change the output of a basic block. We observe that the output of a basic block usually contains multiple variables whose data dependency graphs within the basic block often overlap. In Figure 4, regions with different colors corresponding to the variables a, b, and n represent their data dependency graphs. To ensure that the optimization to the data dependencies of one variable does not affect the result of other variables, the compiler usually considers optimizing the overlapped and independent parts of the DDG, respectively. Therefore, we consider that the corresponding parts in DDG of *HIR* and *LIR* has the same semantic. Based on this observation, we design an Optimal Translation Unit (OTU) to divide the overlapping and independent parts of each dependency path of the basic block, which consists of two steps.

Step 1: *OTU* divides a basic block into multiple non-overlapping units. Starting from the input variables of the basic block, *OTU* traverses the entire DDG of the basic block and marks all instructions with two or more out edges as unit boundaries. The *OTU* obtains independent non-overlapping units based on the boundaries. After that, a DDG between units (UDG) can be constructed according to the data dependencies between statements. As shown in Figure 5 (a), the *OTU* first divides the DDG of *LIR* into five non-overlapping units. Each unit is regarded as a node of UDG, and the dotted lines



(a) DDG_l (b) DDG_l after combined (c) DDG_h

Figure 5: An example of pair matching of OTU

are dependency edges. In the same way, Figure 5 (c) generates a UDG of *HIR*.

Step 2: *OTU* partially merges the units divided in Step 1, because there are units whose out edges all point to one unit. For example, there are 2 out edges between TU1a and TU1b in Figure 5 (a). This is due to compiler optimizations, such as the division optimization in Figure 5 that causes operations on a variable to be divided into 3 units TU1a, TU1b, and TU1c. We iteratively combine such units until no unit in UDG has all the out edges pointing to the same unit. For example, in Figure 5, (b) is the result of the merging of (a).

Training Dataset. Since the neural model requires labeled data in the training phase, we use *OTU* to partition both the basic blocks of *LIR* and *HIR* when obtaining the training set. Due to the optimization strategies of the compiler, the CFG of *HIR* often does not match precisely with the CFG of LPL or *LIR*. Inaccurate matching between the basic blocks will lead to inaccurate labeling of the training set. To avoid this problem, we choose functions that contain only one basic block and then segment them to form the training set of *NeurDP* (see Section 4). Based on our observation, optimization across basic blocks only changes the segmentation and their orders, which does not introduce new types of instructions and mappings. Therefore, the model trained on our dataset can accurately translate the *LIR* instructions in each basic block of a complex function to the corresponding *HIR* instructions. Existing rule-based decompilers [4, 6, 21] are also implemented based on this principle.

After applying *OTU* in both *LIR* and *HIR*, we try to map units of them to label dataset. We observe that their UDGs are usually isomorphic, although the DDGs of *LIR* and *HIR* may be different. Therefore, we match UDGs of *LIR* and *HIR* to pair their units and get labeled. If two nodes in UDG cannot be distinguished, we first examine their internal instructions and distinguish them by some special features, including their constants, const strings, and the address of a procedure call. For nodes that are still indistinguishable according to these features, we throw them away. We use this labeling method to ensure the accuracy of labels.

3.3 Neural Translation

In the field of neural translation, sequence-to-sequence (seq2seq) neural networks [8, 18, 32] have achieved excellent results and have been applied in commercial products such as Google Translate [33]. Therefore, previously studies (e.g., TraFix [20], Coda [16], and Neutron [24]) utilize such models for translation. However, these existing neural machine decompilers do not work well, especially for the optimized LPL, indicating that the seq2seq models cannot effectively cope with the decompilation tasks of LPL. The low accuracy is mainly because seq2seq neural networks do not consider the data dependencies between instructions, which is vital for the compiler to generate LPL. For example, in Figure 6 (c), the seq2seq neural networks view the *LIR* as the sequence `<smull, lsr, add3, asr, add5, lsl>`, but ignore the data dependencies (e.g., `<smull→lsr>` and `lsr→add3>`). So the model wrongly decompiles the result as shifting and arithmetic operations. However, the correct result is the division operation `div`.

Based on the above observation, the neural network model should capture the instructions and the data dependencies between instructions. So we choose to use Graph Neural Networks (GNN). It can capture the features of nodes (i.e., instructions) and edges (i.e., data dependencies) in DDG. Thus, the model's decompilation problem can be defined as follows: Given the *LIR*'s DDG subgraph G as input, the model outputs the corresponding *HIR* sequence, expressed as $P(Y) = P(Y|G)$.

We adopt the graph-based neural network gated graph sequence neural network (GGS-NN) [23]. We do not show the details of the model here. Figure 7 shows the model's architecture (i.e., an encoder-decoder architecture). The encoder uses the gated graph neural network (GG-NN) [23]. The node initialization module aims to define the initial state of the node and perform initialization operations on the DDG. The decoder uses a long short-term memory (LSTM) network with the bridge mechanism. Considering the inputs are *LIR*/*HIR* units which are not complicated, we use a 2-layer LSTM network. The global attention [26] is introduced to improve the model's performance. The token embedding [27] module is used to generate the word vector of the output sequence, and we use the learnable multi-dimensional embedding vector. Regarding the loss function, we use the Kullback-Leibler divergence [22] as follows.

$$D_{KL}(p||q) = \sum_{i=1}^{n} p(x) log \frac{p(x)}{q(x)} \quad (1)$$

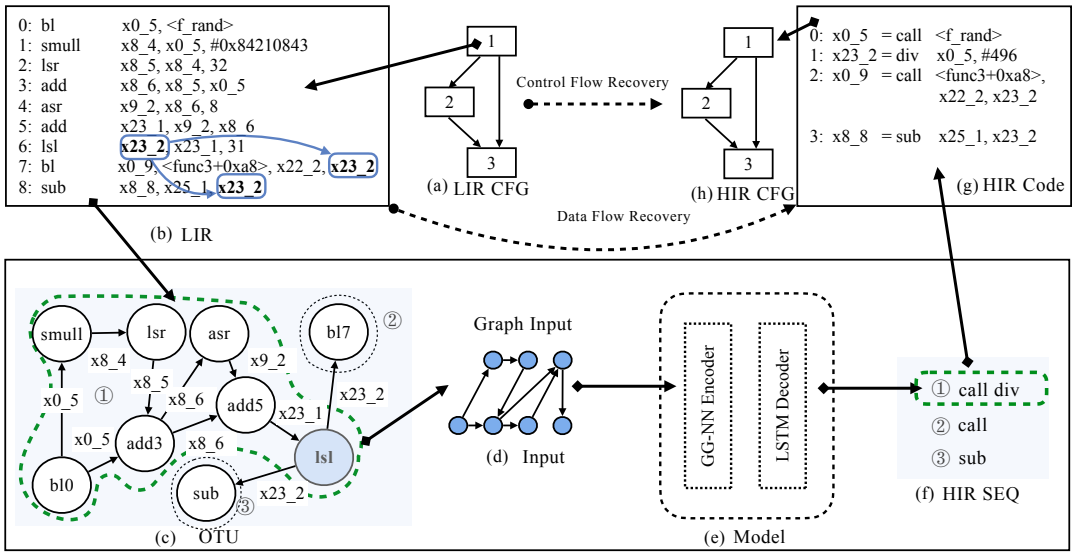


Figure 6: Neural translation process

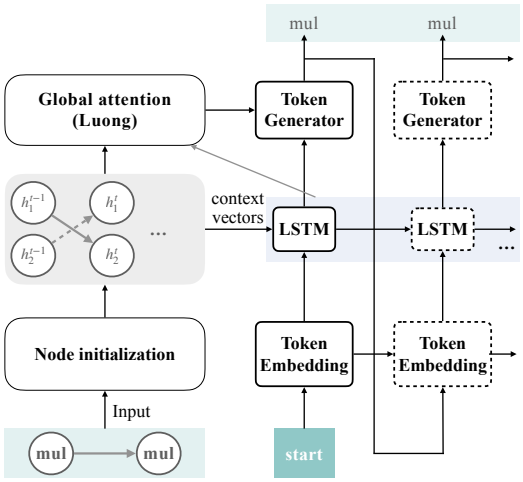


Figure 7: Model architecture

Based on our evaluation, our model is much more accurate (29.58% higher on average) than seq2seq neural networks. We further look into the code and find that GNN correctly captures the features of data dependencies. For the example in Figure 6, our model can correctly decompile the *LIR* to `div`, which means our model is effective even for optimized code.

3.4 Operands Recovery

Recall that the output of our model does not contain real operands. To accurately recover the *HIR* operands, we further split the *LIR* unit, pair *LIR*/*HIR* instructions, and recover operands in each unit. In this step, we use operands in LIR to fill into the HIR. We first pair the instructions with the same semantic meanings, which can be obtained by analyzing the instructions manually. For example, in Figure 6, the *LIR* instruction `bl` has the same semantic meaning as the *HIR* instruction `call`. So we pair them together. Note that identifying the semantic meaning of instructions is only a one-time effort. Then, for the unpaired instructions, we pair them in the order of their addresses. For example, in Figure 6, the *LIR* instructions between Line 1 and Line 6 are paired to the *HIR* instruction `div`.

After obtaining the instruction pairs, we design a data-flow-based approach to recover the *HIR* operands. For each pair, we identify the destination operand in *LIR* (from the node that has no output link in DDG) and use it as the destination operand in the *HIR* instruction. Then we identify the source operands in *LIR* (from the node that has no parents in DDG) and put them as the source operands in the corresponding *HIR* instruction. For some special instructions in *HIR* and *LIR* (e.g., `div` and `madd`), we make rules to find the source operands and the destination operands. For example, in Figure 6, we map instructions 1-6 (18 operands) in the *LIR* to instruction 1 (3 operands) in the *HIR*. By analyzing the DDG of *LIR*, we get the output variable $x23\_2$, input variable $x0\_5$, and 4 immediate #0$x84210843$, 32, 8, and 31, which are not defined inside the DDG. Operands $x23\_2$ and $x0\_5$ can be assigned to *HIR* to the corresponding position. Besides, we manually make the division optimization rules to get another operand #496.

4 EVALUATION

In this section, we describe our experiments to evaluate *NeurDP*'s performance. Firstly, we evaluate the accuracy of decompilation tools at different optimization levels, which can reflect the ability of each decompiler to respond to the optimization strategies related to the expression and data flow in the compiler optimization. We compare *NeurDP* with two state-of-the-art neural-based decompilers [16, 24]. To evaluate the efficiency of our model and *OTU*, we

compare *NeurDP* with other baseline models and other methods of splitting basic blocks. We also analyze the decompiling results of one famous open-source decompilation tool RetDec [21].

4.1 Experiment Setup

Dataset. To build the dataset, we randomly generated 20,000 functions, consisting of arithmetic and calling statements, compiled them using *clang10.0* with optimization levels `O0` to `O3`. By capturing the intermediate results and reversing the binaries, we got 80,000 *LIR*/*HIR* pairs. Then, we use *OTU* to split the function into smaller units for training. After removing the duplicated units and batches that were not full, we got 242,000 pairs. We randomly selected 220,000 pairs for training, and the rest 22,000 pairs were used for validation. We evaluate *NeurDP* from the following aspects: accuracy and generalizability.

Platform. All our experiments are conducted on a 64-bit server running Ubuntu 18.04 with 16 cores (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz), 128GB memory, 2TB hard drive and 2 GTX Titan-V GPU.

4.2 Accuracy

Metrics. As mentioned above, the compiler's optimization changes the statements in the source code. So the decompiled code may not be the same as the original source code at the statements level, even if both codes have the same functionality. We propose a method to evaluate the compiler's accuracy in solving this problem. We consider the decompiled basic block correct if the semantics of this HPL's basic block is the same as the semantics of the corresponding LPL's basic block. Below, we will introduce our comparison method for the semantics of two basic blocks.

Basic block can be abstracted to a function F, mapping the input set $IN(B)$ to the output set $OUT(B)$. We define them as follows: $IN(B) = \{in_0, in_1, ..., in_n\}$, where in_i is the i-th input of a basic block. $OUT(B) = \{out_0, out_1, ..., out_n\}$, where out_i is the i-th output of a basic block. $F = \{f_0, f_1, ..., f_n\}$, where f_i is the i-th function of a basic block, $out_i = f_i(IN(B))$, $OUT(B) = F(IN(B))$. We define the accuracy of a basic block as $Acc_B = Count_{correct}(f_i^{HPL})/Count(F^{LPL})$, where f_i^{HPL} is the i-th function of HPL. To find the correct f_i, we first locate the corresponding out_i^{HPL} in the decompiled HPL for each ${out_i}^{LPL}$ in LPL (e.g., pointer to the same variable). For *NeurDP*, it is easy to determine whether the two outputs correspond or not since we map the variables in *LIR* directly to *HIR* when recovering the variables in Section 3. For other decompilers, we pair outputs by manual analysis. Then we can get corresponding $< out_i^{HPL}, out_i^{LPL} >$ pairs. Given $IN(B)$, we consider the f_i^{HPL} obtained by decompiling to be correct if the results of the function f_i^{HPL} and f_i^{LPL} are equal for each paired out_i^{HPL} and out_i^{LPL}. At last, we define the *program accuracy* as $Acc = \sum_{k=1}^{N} Acc_B(B_i)/N$, where N is the number of basic blocks in the program, B_i is the i-th basic block in the program. In the evaluation, *NeurDP* automatically generates functions from the basic blocks, and we manually check if f_i^{HPL} is correct by comparing the functions from HPL and LPL. For example, in Figure 1, HPL and LPL codes of `func1` all contain one basic block. We can get the accuracy of `func1` $Acc = Acc_B(B)/1$. The output set and input set of B in HPL are $\{ret_{HPL}\}$ and $\{p0\}$. The output set and input set of B in LPL are $\{ret_{LPL}\}$ and $\{w19\}$. Then, we can get output pairs in HPL and LPL $\{< ret_{HPL}, ret_{LPL} >\}$. The corresponding function of ret_{HPL} is $f = (f\_scanf\_nop() + p0 \times f\_rand())/f\_scanf\_nop()/(-123)$. And the corresponding function of ret_{LPL} is $f = (f\_scanf\_nop() + w19 \times f\_rand())/f\_scanf\_nop()/(-123)$. Note that here we should make rules to handle the division optimization for LPL. We can manually compare these two functions. At last, we can get the accuracy of `func1` $Acc = Acc_B(B)/1 = (1/1)/1 = 1$, where $Count_{correct}(f^{HPL}) = 1$ and $Count(F^{LPL}) = 1$.

Table 4: Results on different compiler-optimization level

Strip option	Compiler optimization level			
	`O0`	`O1`	`O2`	`O3`
no	0.95	1	0.9	0.9
debug	0.95	0.92	0.9	0.9
all	0.95	0.92	0.9	0.9

`no`: remain all symbolic information.
`debug`: strip debug information.
`all`: strip all symbolic information.

Table 5: Comparison with state-of-the-arts

	Compiler optimization level			
	`O0`	`O1`	`O2`	`O3`
Neutron	87.78%	35.45%	32.79%	32.81%
Coda	67.2%-89.2%*	-	-	-
RetDec	29%	25%	4%	-
NeurDP	**95**%	**94.67**%	**90**%	**90**%

*Program accuracy 67.2%-89.2% of Coda is from Table 2 in [16] .

Settings. To evaluate the accuracy of *NeurDP*, We develop a tool using cfile [2] to randomly generate 1,000 pieces of code as *DS1*, and then use Clang to compile them at optimization level `O0`-`O3` to get 4,000 executable and linkable format files (ELF), where each ELF contains 5 functions. Our dataset includes arithmetic expressions, procedure calls, etc. We evaluate the robustness of the decompilation tools through their performance in decompiling binary files with (or without) symbolic information, and different optimization levels. Further, we perform strip [1] operations on the binary code in the above dataset, where *strip debug* refers to removing the debugging information from the binary, *strip all* means removing all symbolic information from the binary code.

Accuracy of NeurDP. Table 4 shows the accuracy of *NeurDP* for four optimization levels from `O0`-`O3`. *NeurDP* can achieve 92.42% accuracy on average at `O0`-`O3` optimization level. It can be seen that the accuracy of *NeurDP* is higher under levels `O0` and `O1`, while it is lower at levels `O2` and `O3`. At `O2` and `O3`, there are more optimizations of the compiler's backend, and the model is more difficult to learn the rules. When the optimization level is increased, the accuracy of the disassembly will reduce, which affects the accuracy of decompilation. Under the same optimization level, there are a few differences in the model's performance with and without symbolic information, which indicates that the model has good robustness to the stripped binaries. Under level `O1`, the accuracy rates of binaries without debug information and any symbol tables are slightly lower than with symbolic information. After the analysis, we found that the disassembly accuracy without symbolic information reduces, and some function boundary recognition errors occurred, leading to unsatisfactory decompilation results.

Comparison with State-of-the-arts. We compare *NeurDP* with the state-of-the-art neural-based decompilers (i.e., Coda [16] and

Table 6: Token accuracy of different neural networks

	Compiler optimization level			
	O0	O1	O2	O3
Transformer-SRC	52.93%	38.22%	39.84%	33.37%
Transformer-AST	75.79%	45.62%	44.60%	32.98%
Transformer-IR	77.18%	75.66%	74.49%	73.94%
LSTM-IR	85.40%	86.61%	85.63%	85.95%
GRU-IR	26.56%	21.93%	25.85%	24.19%
NeurDP	**89.50**%	**93.16**%	**93.40**%	**90.08**%

Neutron [24]). From the results, we see that *NeurDP* outperforms both of them. We do not have access to the source code and dataset of Coda [16]. We also find that the details needed to reproduce Coda are not described in their paper. So we could neither test Coda on our dataset nor test our *NeurDP* on their dataset. Considering that the benchmark ($Math + NE$) [16] in Coda is generated similarly to our dataset, we directly compare the effect with that described in their paper. Coda splits the long and short sentences in the dataset into two groups for testing ($Math{+}NE)_S$ and ($Math{+}NE)_L$. Therefore, the range of Coda's accuracy on these two datasets is listed in Table 5. In addition, since Coda cannot handle the compiler-optimized LPL, this part of the data is replaced with blanks. For Neutron, we get the code and test it on our dataset. The result in Table 5 shows that Neutron does not perform as well as *NeurDP* on our dataset, especially for compiler-optimized code. We further analyze the experimental results, where *NeurDP* adopts *HIR* as the model's target to cope with compiler optimization and uses the *OTU* mechanism to divide the basic blocks into finer-grained partitions. In contrast, the previous neural-based work utilizes HPL or AST as the model's target. Source code and AST are not optimized by the compiler front-end and cannot correspond well with the optimized LPL, increasing the difficulty of model learning. Therefore, Coda and Neutron cannot cope with the optimized code very well.

Compare with Different Neural Networks. To understand the effect of the GGS-NN model, we compare *NeurDP* with other models. We select three wildly used seq2seq models (Transformer [32], LSTM [18], and GRU [8]) to make a comparison with our *NeurDP*. Note that, instead of using a tree decoder, we serialize (traverse) the AST of the source code as the output of the transformer for model Transformer (AST) in Table 6. We use the same data set as *NeurDP* to train these models separately. Note that we use Clang to extract `<assembly, source code/AST/IR>` as ground truth for these models. In this experiment, we use the accuracy of tokens to evaluate these models. This is because outputs of other models often have so many syntax errors that it is hard to evaluate their functionality. The experimental result proves that the GGS-NN can make decompilation results more accurate (see Section 3.3).

We further evaluate the effectiveness of the NMT model using *HIR* as a translation target. We choose the Transformer model as the baseline and use source code, AST, and *HIR* as the model's translation targets for evaluation on DS1. From the first three lines of Table 6, we can find that when the NMT model uses source code (SRC) or AST as the translation target (output), its effect on `O1-O3` is far worse than that at `O0`. In contrast, the model that uses *HIR* as the output has no significant difference in translation effects at the `O0-O3`, and the complete accuracy is better than the other two models. The experimental results show that using *HIR* as the translation target of the model is highly generalizable for optimized code, which are not affected by compiler optimizations. What's more, using our *HIR* and *LIR* pairs splitting by *OTU* performs better than other models.

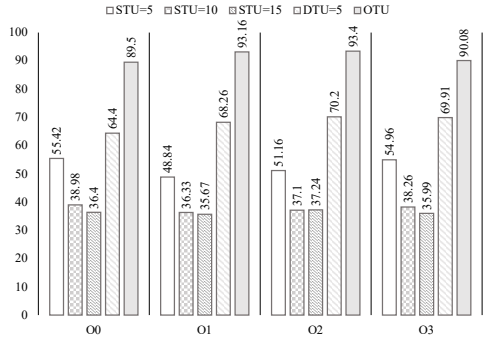


Figure 8: Token accuracy under different forms of TU

Impact under Different Translation Unit. We evaluate the token accuracy of using different methods of splitting basic blocks, including code sequence-oriented *TU* (STU) and DDG-oriented *TU* (DTU). We select *DS1* to evaluate the performance of different forms of TU. To verify the effect of *TU*, we choose the statement size of the *STU* as 5, 10, and 15 on the assembly sequence. To further verify the performance between fixed-length and variable-length *DTU*, we select a fixed-length *DTU* with a size of 5 and our *OTU*. Figure 8 shows the performance of *NeurDP* under different forms of *TU*. The result indicates that *NeurDP* becomes less effective in code sequences as the *STU* increases, mainly because the longer the code the model needs to handle, the more difficult it is to translate accurately. If we do not split the basic block, the results will worsen. In addition, we find that the fixed length of *TU*, either sequence-oriented or DDG-oriented, makes the correspondence between *LIR* and *HIR* in the training set more ambiguous, which leads to the model's failure to learn the mapping rules. Compared with the above methods, our *OTU* can maximize the automation of obtaining *LIR* and *HIR* pairs with the correct correspondence for training, thus enabling the model to learn the mapping relationship between them quickly and accurately. What is more worth mentioning is that our approach can deal with compiler optimization problems well.

Analysis of Rule-based Decompilers. We also evaluate the performance of one famous open-source rule-based decompiler RetDec [21], which contains over 100,000 lines of code and is a representative decompilation work. In the evaluation, RetDec does not perform as well as those three neural-based decompilers (only achieving 29%) on unoptimized binaries. When evaluated on `O1`, RetDec achieves 29% accuracy without debug information and 17% accuracy without any symbolic information. When evaluated on `O2` and `O3`, RetDec achieves only 4% accuracy because mostly binaries are failed to decompile. We manually analyzed these samples of decompilation failures and found that many errors occurred in the disassembly step due to the lack of symbolic information. The code sections could not be accurately located. In some cases, although the function entry point was found, the decompilation is broken due to some small disassembly errors. Moreover, we studied the mechanism of rule-based decompilers [5, 6, 21]. Rule-based decompilation tools have more rules and constraints and are more sensitive to disassembly errors. A memory or stack error in the

assembly will often cause the following code or the entire function to fail to decompile. In contrast, *NeurDP* has no constraints (e.g., stack balance), so it has a certain degree of fault tolerance for some disassembly errors, such as *sp stack unbalanced* caused by inline assembly code. It will not cause the entire decompilation process to fail. Even if the disassembly error leads to the wrong decompilation result, *NeurDP* can still finish decompiling without triggering an error and stopping like other tools.

5 DISCUSSION

Limitations. In this work, we propose and implement a novel neural decompilation approach, named *NeurDP*, to demonstrate that the neural-based approach copes with the decompilation problem of compiler-optimized LPL. However, *NeurDP* still has some limitations. Firstly, the HPL statements generated by *NeurDP* are mapped directly from *HIR* and are mostly monadic or binary statements. For example, for an expression `a = b + c * d`, *NeurDP*'s outputs are two statements `tmp = c * d, a = b + tmp`. Such problems could be solved through data dependency analysis. Besides, the quality of HPL decompiled by *NeurDP* is closely related to the accuracy of the LPL (assemble code). We find that for stripped binaries, the disassembly tools (e.g., RetDec) may cause errors in assembly code due to incorrect function boundary identification, especially for decompiled code, which affects the quality of our *NeurDP*'s performance. Secondly, *NeurDP* is not completely end-to-end from LPL to HPL. The lifting from IR to HPL depends on the rules, so it is not easy to support multi-machine and multi-language. Thirdly, *NeurDP* is a prototype system, and the dataset contains only the statements consisting of arithmetic and calling operations to integer variables. We plan to include more types of statements in future work. Finally, *NeurDP* directly uses the existing techniques, including disassembly, reconstruction of control structures, etc. *NeurDP* does not consider the errors introduced by these modules so these components will affect the final accuracy.

Future Work. We will continue to explore techniques for improving the decompilation quality of *NeurDP* and resolve the above limitations. For example, we will eliminate some binary statements by merging expressions and reducing the redundant variables for HPL generated by *NeurDP*. The elimination of sentences will be achieved through data-flow analysis. At the same time, we will use neural-based methods to learn some patterns of statements that match developers' habits through historical experience and guide the process of merging to generate HPL that is more in line with programming habits. Furthermore, we will combine the collective capability of the-state-of-art commercial and open-source disassembly tools [10, 15], to generate high-quality assembly code. The goal is to ensure the *NeurDP*'s input is correct, which is a sufficient condition to ensure the performance of the decompiled code.

6 RELATED WORK

Rule-based Decompilation. Rule-based decompilation techniques rely on PL experts to customize and design specific heuristic rules lifting low-level PL to high-level PL and achieving software decompilation. The current popular rule-based decompilation tools are Hex-Rays [4], RetDec [21] and Ghidra [6]. Hex-Rays is a decompiler engine integrated into the commercial reverse tool IDA Pro, which is the de-facto industry standard in the software security industry. However, Hex-Rays is not open-source, and the inside technology is hard to understand. RetDec is an LLVM-based redirectable open-source decompiler developed by Avast in 2017, aiming to be the first "universal" decompiler that can support multiple architectures and languages. RetDec can be used alone or as a plug-in to assist IDA Pro. Ghidra is an SRE framework developed by the National Security Agency (NSA) for cybersecurity missions. Ghidra supports running on Windows, macOS, and Linux, supporting multiple processor instruction sets and executable formats. Although these studies have made significant improvements, they are far from perfect. Rule-based approaches are needed to manually detect known control flow structures based on written rules and patterns. These rules are difficult to develop, error-prone, usually only capture part of the known CFG, and require long development cycles. Worse still, these methods do not work well when decompiling optimized code. However, optimization is now the default option when commercial software is compiled. Unlike rule-based decompilers, the goal of *NeurDP* is based on deep neural networks to learn and extract rules from code data automatically. Trying to break through the problem of compiler-optimized code is challenging to decompile accurately.

Learning-based Decompilation. Most of the existing learn-based decompilation methods [16, 19, 20, 24] draw on the idea of NMT to transform the decompilation into the problem of mutual translation of two different PL. Katz et al. [19] first proposed an RNN–based method for decompiling binary code snippets, demonstrating the feasibility of using NMT for decompilation tasks. Katz et al. [20] proposed a decompilation architecture based on LSTM called TraFix, and they realized that the primary task of building a decompilation tool based on NMT is to make up for the information asymmetry between high-level PL and low-level PL. TraFix takes the preprocessed assembly language as input and the subsequent traversed form of C as output, which reduces the structural asymmetry between the two PLs. Fu et al. [16] proposed an end-to-end neural decompilation framework, named Coda, based on several neural networks and different models used for different statement types. Coda [16] can accurately decompile some simple operations, such as binary operations, which is far from actual application. Unlike the above existing studies, our neural decompilation framework *NeurDP* can decompile the real-world low-level PL code, especially the compiler-optimized PL code, into a C-like high-level PL code with corresponding functionality.

7 CONCLUSIONS

In this paper, we propose and implement a neural decompilation framework named *NeurDP*, which accurately decompiles LPL code to a C-like HPL with similar functionality. We also design an optimal translation unit (OTU) suitable to form a dataset for learning algorithms better capturing the relationship between HPL and LPL. The evaluation results show that *NeurDP* achieves better accuracy for optimized code, even compared with the state-of-the-art.

ACKNOWLEDGMENTS

This work was supported by NSFC U1836211, Beijing Natural Science Foundation (No.M22004), Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

[1] 2009. strip. https://linux.die.net/man/1/strip.
[2] 2021. cfile. https://github.com/cogu/cfile.
[3] 2021. Github Copilot. https://copilot.github.com/.
[4] 2021. Hex-Rays. https://www.hex-rays.com/products/decompiler/.
[5] 2022. Decompiler and Beyond. https://infocon.org/cons/.
[6] 2022. Ghidra. https://ghidra-sre.org/.
[7] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*. 353–368.
[8] 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. *arXiv preprint arXiv:1406.1078* (2014).
[9] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 99–116. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua
[10] Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. 2021. DisCo: Combining Disassemblers for Improved Performance. (2021).
[11] Steven HH Ding, Benjamin CM 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)*. IEEE, 472–489.
[12] Evan Downing, Yisroel Mirsky, Kyuhong Park, and Wenke Lee. 2021. {DeepReflect}: Discovering Malicious Functionality through Binary Reconstruction. In *30th USENIX Security Symposium (USENIX Security 21)*. 3469–3486.
[13] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. {FIRMSCOPE}: Automatic uncovering of privilege-escalation vulnerabilities in pre-installed apps in android firmware. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2379–2396.
[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[15] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. 1075–1092.
[16] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*. 3703–3714.
[17] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. 2020. Bigmac: Fine-grained policy analysis of android firmware. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 271–287.
[18] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
[19] Deborah S Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 346–356.
[20] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards Neural Decompilation. *CoRR* abs/1905.08325 (2019). arXiv:1905.08325 http://arxiv.org/abs/1905.08325
[21] Jakub Křoustek, Peter Matula, and P Zemek. 2017. Retdec: An open-source machine-code decompiler.
[22] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics* 22, 1 (1951), 79–86.
[23] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated Graph Sequence Neural Networks. *arXiv e-prints*, Article arXiv:1511.05493 (Nov. 2015), arXiv:1511.05493 pages. arXiv:1511.05493 [cs.LG]
[24] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. 2021. Neutron: an attention-based neural decompiler. *Cybersecurity* 4, 1 (2021), 1–13.
[25] Zhibo Liu and Shuai Wang. 2020. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 475–487.
[26] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).
[27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
[28] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2020. Xda: Accurate, robust disassembly with transfer learning. *arXiv preprint arXiv:2010.00770* (2020).
[29] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
[30] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. 2018. Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 273–290.
[31] Michael James Van Emmerik. 2007. *Static single assignment for decompilation*. University of Queensland.
[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 5998–6008. http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf
[33] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). http://arxiv.org/abs/1609.08144
[34] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 158–177.
[35] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS*. Citeseer.