Beijing Forest Studio

北京理工大学信息系统及安全对抗实验中心

二进制代码反编译技术研究

硕士研究生 高玺凯

2025年04月06日

问题回溯

- 总结反思
  - 讲解语速过快
  - 论文算法原理分析不够深入

- 相关内容
  - 2025.03.02 徐程柯《二进制代码补丁存在性测试》
  - 2024.10.08 高玺凯《二进制代码相似性检测技术》

内容提要

- 预期收获
- 内容引入
- 内涵解析与研究目标
- 研究背景与意义
- 研究历史与现状
- 知识基础
- 算法原理
  - ReF Decompile
  - DeGPT
- 特点总结与未来展望
- 参考文献

预期收获

- 预期收获
  - 掌握二进制代码反编译技术的基本概念
  - 了解二进制代码反编译技术的研究背景和研究意义
  - 明确二进制代码反编译技术的前沿方法和未来发展

内容引入

编译

汇编

```int func0(float num[], int size,
  float threshold) {
  int i, j;
  for (i = 0; i < size; i++)
    for (j = i + 1; j < size; j++)
      if (fabs(num[i] - num[j])
          < threshold)
          return 1;
  return 0;}
```

源代码
（面向人类的）

```
undefined4 func0(float param_1,long param_2,int param_3){
  int local_28;
  int local_24;
  local_24 = 0;
  do {
    local_28 = local_24;
    if (param_3 <= local_24) {
      return 0;}
  while (local_28 = local_28 + 1, local_28 < param_3) {
    if ((double)((ulong)(double)(*(float *)(param_2 + (long)local_24 * 4) -
        *(float *)(param_2 + (long)local_28 * 4)) &
        SUB168(_DAT_00402010,0)) < (double)param_1) {
      return 1;}}
    local_24 = local_24 + 1;
  } while( true );}
```

反编译器生成的伪代码

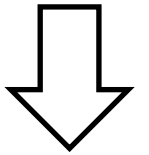
反编译

反汇编

内涵解析与研究目标

- 内涵解析
  - 二进制代码（Binary Code）：由高级编程语言编写的源代码经过编译、汇编、链接后，生成的可在计算机硬件上直接执行的机器码，仅由0和1组成
  - 汇编代码（Assembly code）：介于机器码与源代码之间的低级语言。使用助记符和操作数来描述指令，相较于二进制代码更易读、更易编写
  - 反编译（Decompilation）：将汇编代码自动提升为更高级别且人类可读的高级编程语言代码的过程

```
<func0>:                    ...
endbr64                     mov   $0x0,%eax
push  %rbp                  pop   %rbp
...                         retq
```



```
int func0(float num[], int size,
  float threshold) {
  int i, j;
  for (i = 0; i < size; i++)
    for (j = i + 1; j < size; j++)
      if (fabs(num[i] - num[j])
          < threshold)
          return 1;
  return 0;}
```

- 研究目标
  - 结合程序分析、深度学习等理论
  - 在保持代码语义一致的基础上，提高反编译生成代码的简洁性和可读性，促进对二进制程序快速且准确的理解

研究背景与意义

- 研究背景
  - 随着开源代码和第三方库被广泛复用，已知漏洞在各类计算机系统和物联网设备中迅速传播，带来严重安全隐患
  - 许多关键行业（如航天、轨道交通、电力等）仍在运行仅以二进制形式保存的历史遗留系统，源代码早已遗失或无法维护
  - 大语言模型等人工智能技术的快速发展，为反编译技术提供强有力的技术支撑
- 研究意义
  - 为二进制安全研究提供另一种解决思路，在漏洞检测、恶意软件分析等逆向工程应用中发挥着关键作用，能显著提升软件的安全分析能力
  - 能够帮助开发人员更快速、准确地理解二进制程序，实现二进制程序的修复、维护、迁移和二次开发

研究历史与现状

- 基于规则的传统反编译工具
  - Ghidra、IDA Pro、Binary Ninja…
  - 依赖于手动编写的指令模式、语法结构和控制流/数据流分析等技术，对汇编代码进行结构化解析和语义恢复
  - 局限性
    - 代码可读性不足
      - 冗余结构
      - 无意义标识符
      - 缺乏注释
    - 高度依赖专家经验，难以适应未见模式，泛化能力较弱
    - 规则匹配方法不适用于混淆程序
    - ……

Source Code

```
// Calculate Fibonacci numbers.
int Fibon(int number){

   if (number == 1 || number == 2) {
      return 1;
   } else{
      return Fibon(number - 1) +
 Fibon(number - 2);
   }
}
```

Decompiler Output

```
int Fibon(int param_1) {
   int iVar1;
   int iVar2;

   if ((param_1 == 1) || (param_1 == 2)) {
      iVar2 = 1;
   } else {
      iVar1 = Fibon(param_1 + -1);
      iVar2 = Fibon(param_1 + -2);
      iVar2 = iVar2 + iVar1;
   }
   return iVar2;
}
```

研究历史与现状

2019

Katz等人提出基于神经机器翻译的TraFix方法。利用LSTM生成结构化代码模板，并通过模板填充修正常量以确保语义等价；引入语法标准化和依赖图评估机制，提升了准确率和泛化能力

基于神经机器翻译的方法

2022

Hosseini等人提出无需语言特定知识的BTC方法。使用Transformer模型训练汇编与源代码函数对，避免构建AST或解析语法，提升了对多语言的可迁移性

基于神经机器翻译的方法

2023

Armengol等人提出SLaDe，一种轻量级Transformer神经反编译器。以函数级的汇编–C代码对进行训练，结合无dropout的正则化训练、子词级标记和类型推理模块，在多ISA及多优化级别下均表现优异

端到端方法

2024

Tan等人提出LLM4Decompile，分别采用End2end和Refined策略，是首个开源反编译专用LLM系列。通过优化训练流程（如数据增强、清洗与双阶段训练），显著改善可读性和可执行性

基于精炼的方法

2024

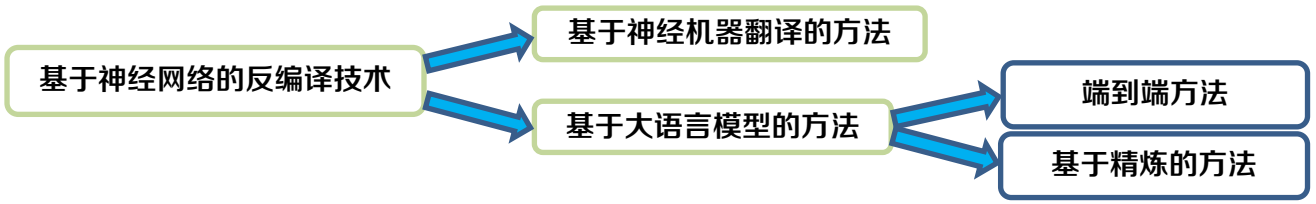
Hu等人提出基于大语言模型的方法DeGPT。通过引入三角色机制，将反编译输出优化任务；并利用MSSC程序分析方法，确保优化不改变原始语义，有效提升了反编译代码的可读性与可执行性

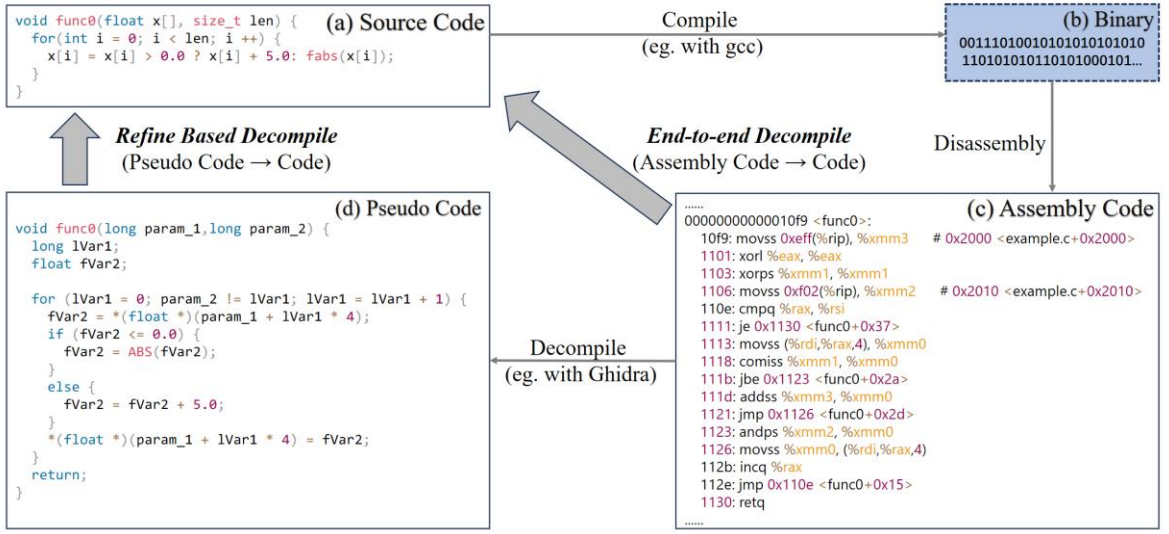
基于精炼的方法

2025

Feng等人提出基于大语言模型的方法ReF Decompile。引入重标记策略保留汇编中的跳转信息，并通过函数调用机制从二进制中提取变量值，解决现有方法在控制流与变量还原上的信息缺失问题，显著增强了反编译结果的准确性与可理解性

端到端方法





端到端方法：直接使用大语言模型从汇编代码重建原始代码
基于精炼的方法：在现有反编译工具生成的伪代码基础上，利用大模型对其进行优化

ReF Decompile: Relabeling and Function Call Enhanced Decompile

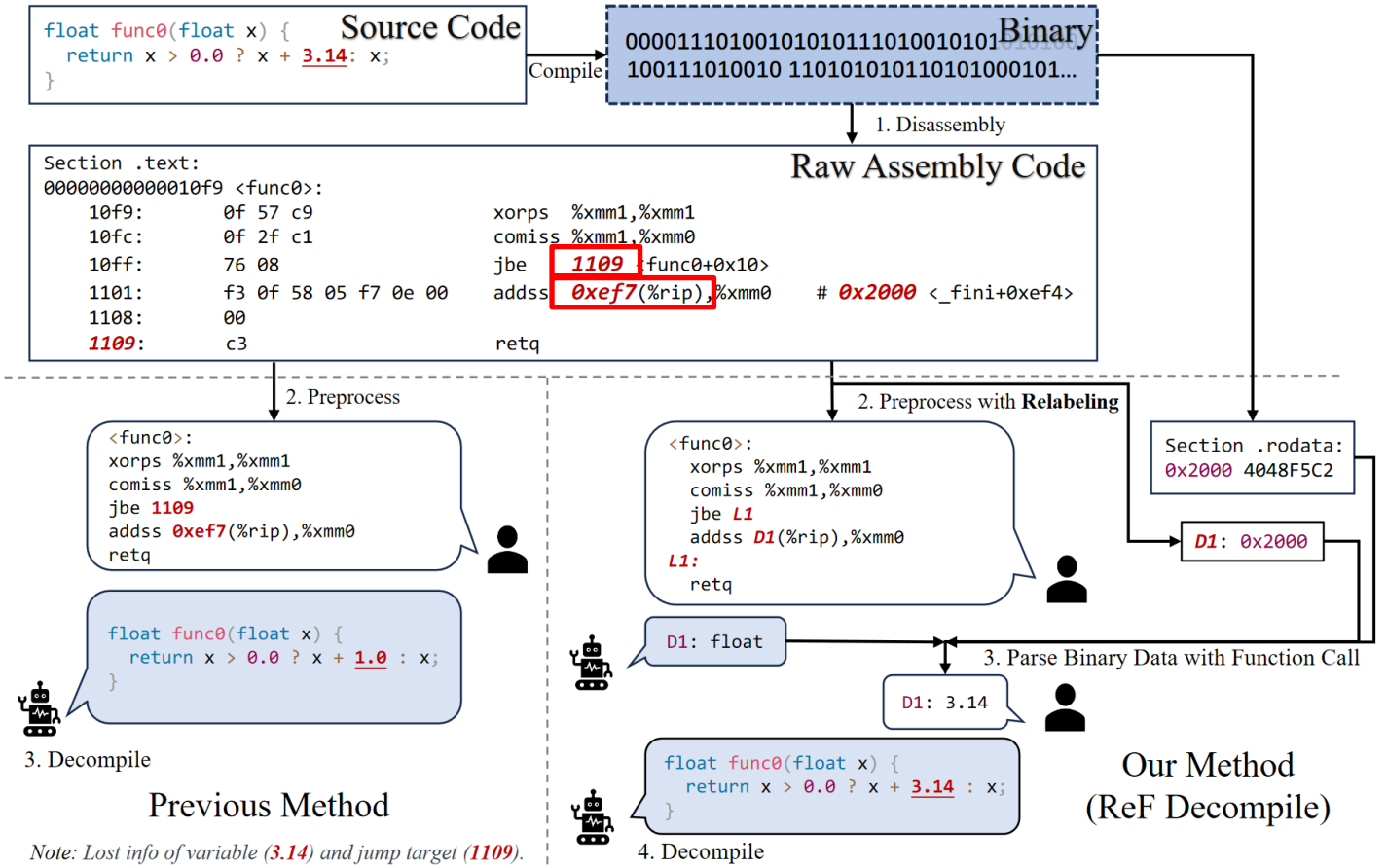
ReF Decompile TIPO

T	目标	从编译后的二进制文件中重构出功能等效的原始源代码，并保证可读性
I	输入	二进制文件*1
P	处理	1. 反汇编 2. 使用重标记策略预处理汇编代码，保留跳转指令的相关信息 3. 处理函数调用请求，获取内存访问指令所引用地址处的数据值 4. 最终反编译
O	输出	反编译生成的源代码*1

P	问题	现有基于大语言模型的端到端方法往往会丢失重构控制流结构和变量所需的关键信息，导致难以准确恢复程序逻辑，限制其准确性和实际应用价值
C	条件	二进制文件未加密/加壳，且未被混淆
D	难点	如何避免预处理过程中控制流结构信息的丢失； 如何获取可执行段外所需的变量信息
L	水平	arXiv 2025

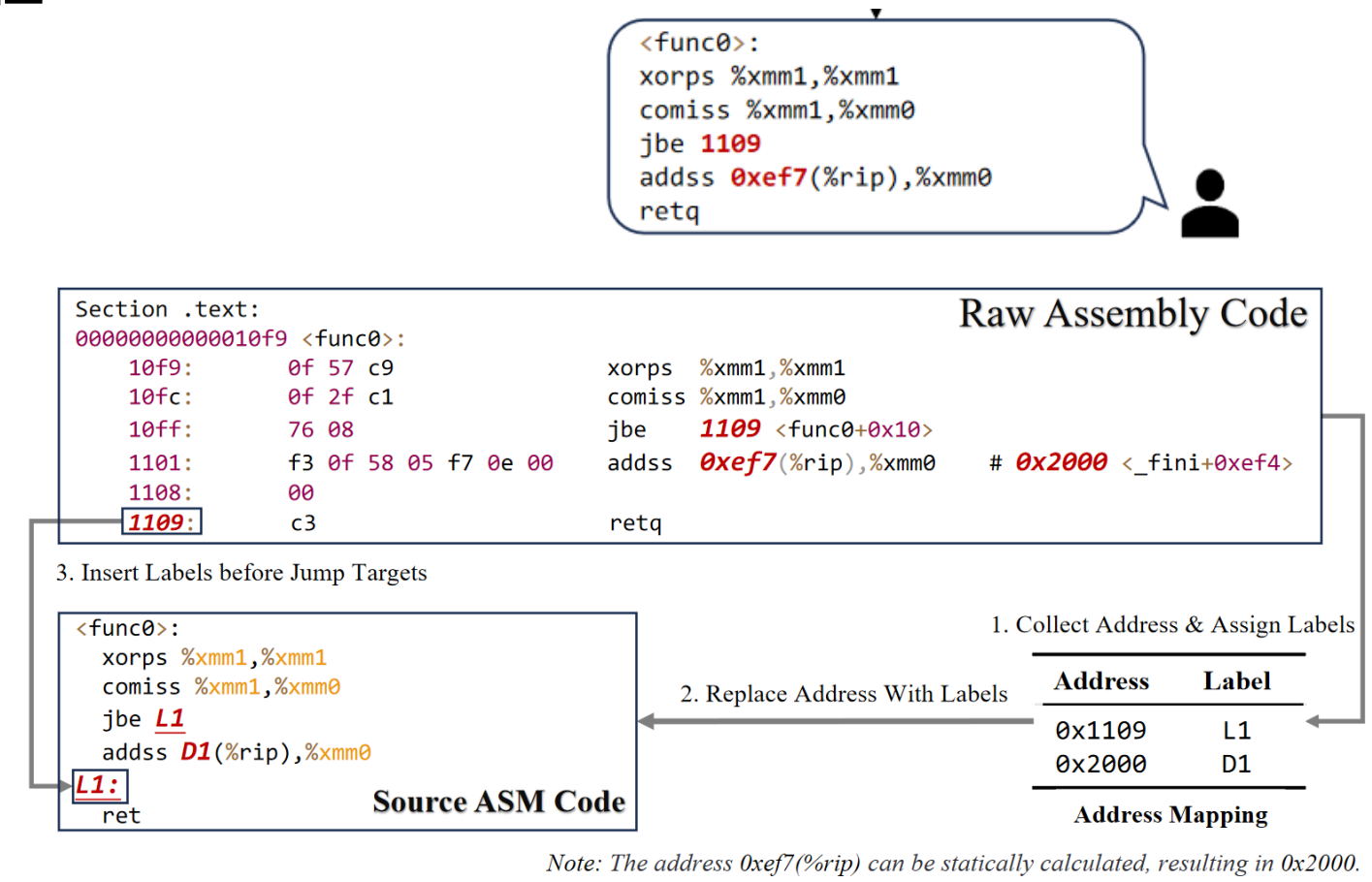
ReF Decompile 算法原理

- 算法原理图
  - 针对问题：现有基于大语言模型的端到端方法往往会丢失重构控制流结构和变量所需的关键信息，导致难以准确恢复程序逻辑
  - 算法流程
    - 反汇编
    - 预处理汇编代码
      - 使用重标记策略，保留跳转指令的相关信息
    - 处理函数调用请求
      - 获取内存访问指令所引用地址处的数据值，补全可执行段外的变量信息
    - 最终反编译



ReF Decompile Relabeling

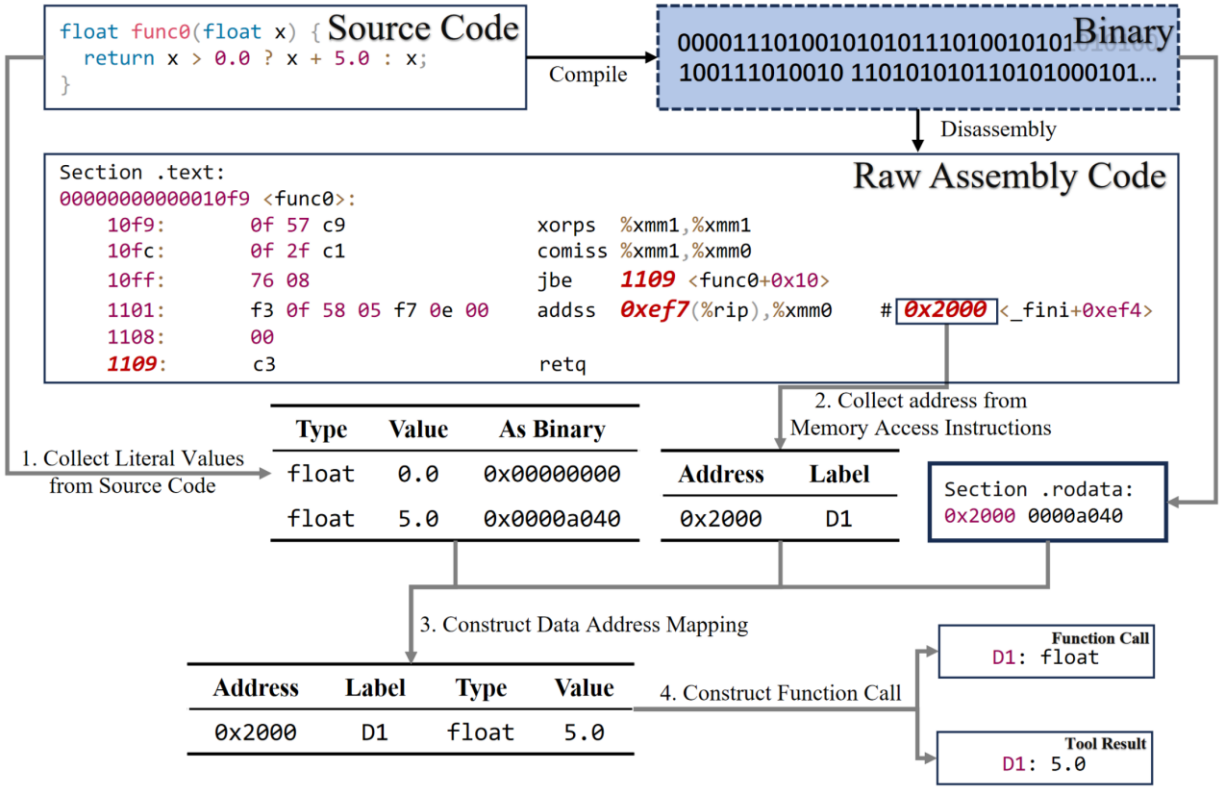
- 现有方法在数据预处理阶段，直接移除地址信息，导致模型恢复准确控制流结构
- 目的：移除汇编代码中的特定地址信息（跳转地址、内存访问地址），同时保留程序跳转逻辑以确保控制流完整性
- 具体步骤
  - 收集地址信息并分配标签
  - 用标签替换特定地址
  - 在跳转目标指令前插入标签
- 通过用直观标签替换特定地址，提升可读性和逻辑清晰度，从而让模型更容易理解跳转逻辑，得到更好的推理结果



Note: The address 0xef7(%rip) can be statically calculated, resulting in 0x2000.

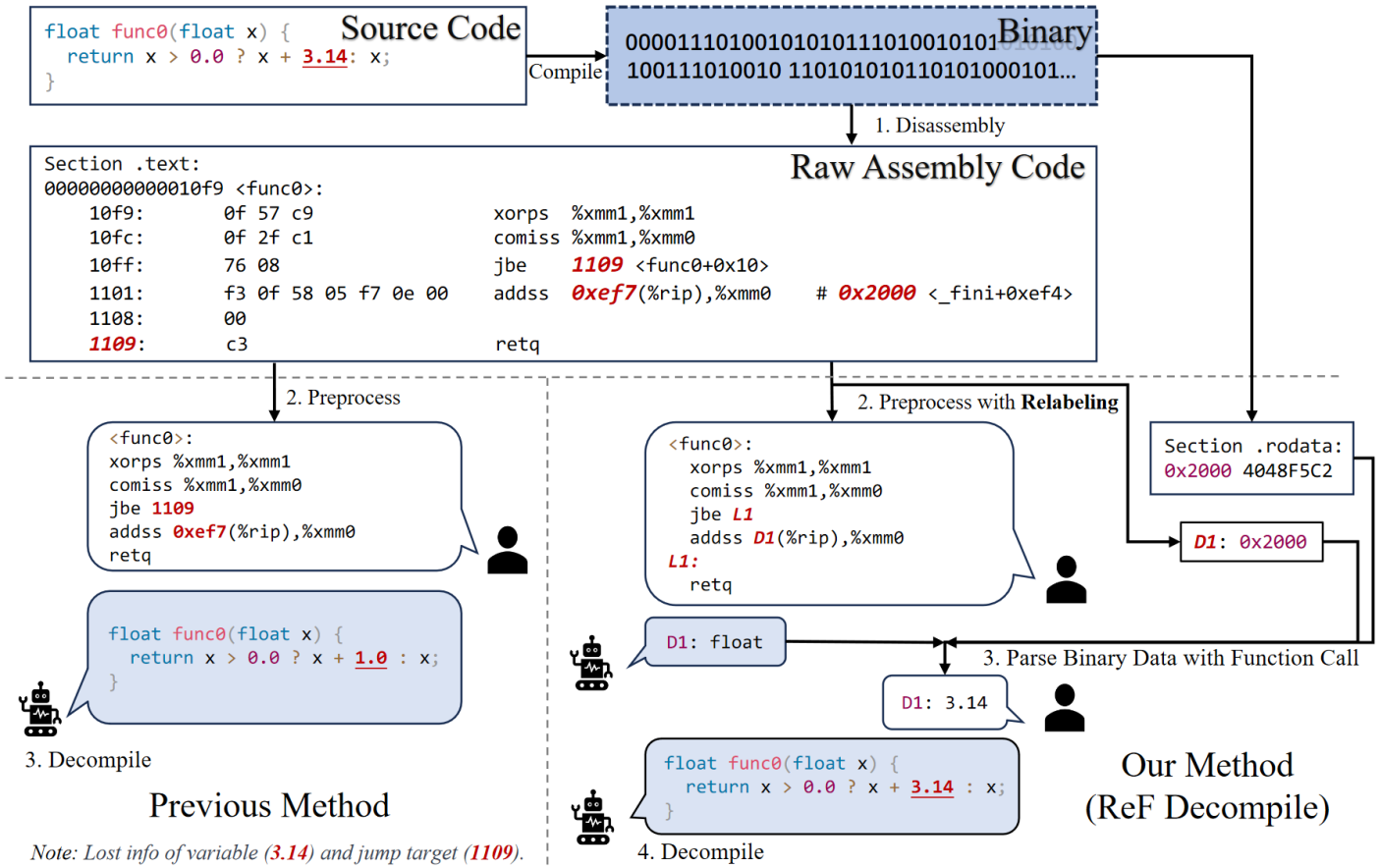
ReF Decompile Function Call 训练阶段

- 现有方法仅依赖于可执行段中的信息，缺失了大量与变量值相关的信息
- 目的：解析源代码并分析二进制文件，以准确提取字面值并将其与二进制文件中的存储地址相匹配
- 具体步骤
  - 提取源代码中的字面值并转换为对应的字节码表示
    - string、float、double等
  - 重标记内存访问地址
  - 构建数据地址映射关系
  - 构建函数调用
    - 调用参数：内存访问标签、数据类型
    - 返回信息：标签对应地址处存储的值



ReF Decompile Function Call 测试阶段

- 模型预测标签对应的数据类型，并通过结构化请求与二进制文件交互，利用标签与地址映射关系，从.rodata段中提取对应数据，并将解析结果反馈回自身，从而补全变量信息



实验设计

- 训练集：Exebench
  - 函数数量 / token数量： 15k / 0.4b
  - 编译器：GCC 11.4
  - 所有函数仅使用标准C库
- 测试集：Decompile-Eval
  - 包含164个编程问题，每个问题附带测试用例
- 基线LLM
  - LLM4Decompile6.7B-End v1.5
- 对比方法
  - 基于规则的传统反编译器：Ghidra
  - 基于精炼的方法：GPT 4o、LLM4Decompile6.7B-Ref
  - 端到端的方法：LLM4Decompile6.7B-End、FAE Decompile 6.7B

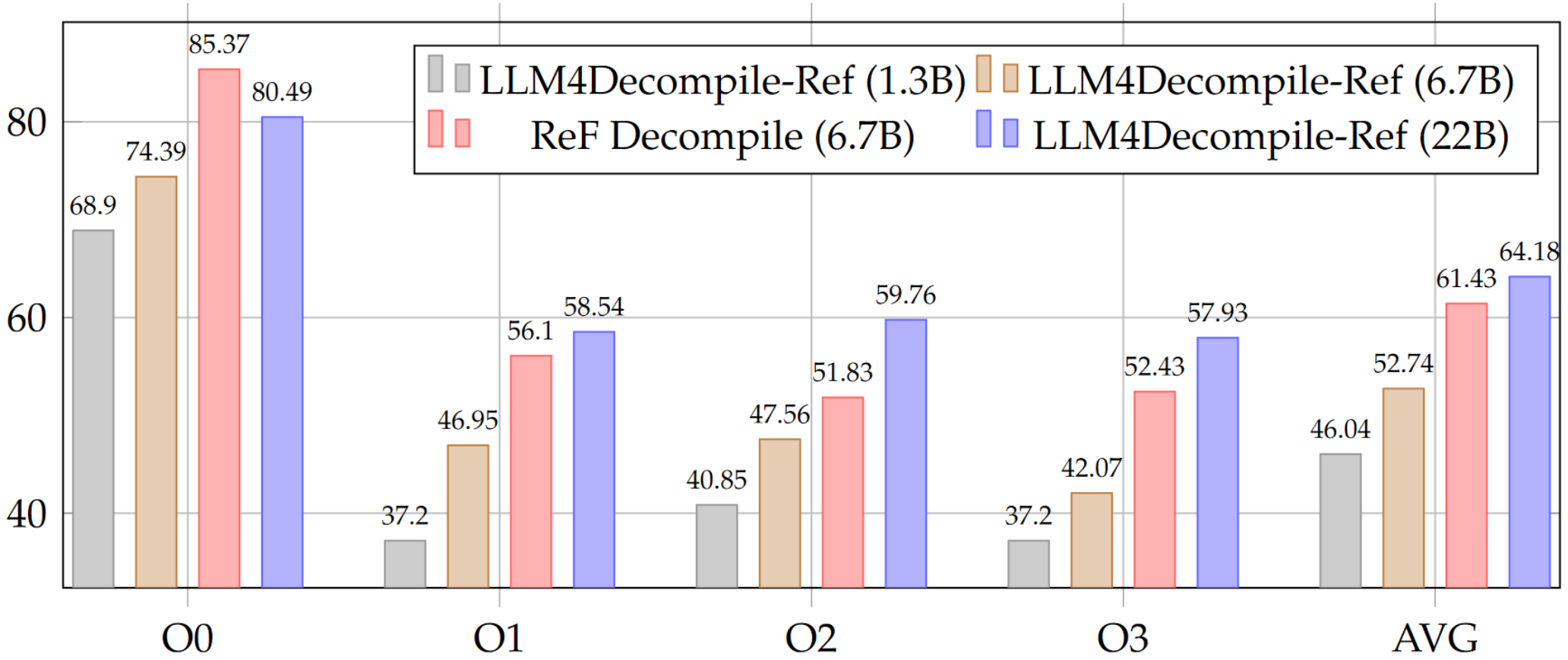
实验设计

- 评价指标
  - 可重执行率（Re-executability Rate）
    - 定义：反编译代码在重新编译并执行后，通过所有测试用例的样本数占所有样本数的比例
    - 评估反编译代码的功能正确性，全面反映反编译代码是否保留原始程序的逻辑行为完整性，是评估反编译工具核心能力的基础性指标
  - 可读性（Readability）
    - 定义：使用GPT-4o并基于一个结构化模板，对变量、循环、条件等语法方面以及逻辑流程与整体结构进行综合评价，并对照原始代码与反编译结果的差异，给出1（差）到5（优秀）之间的分数
    - 评估反编译代码的语义可理解性，衡量其是否接近人工编写代码的维护标准。该指标反映模型对高级语义信息（变量命名、控制流结构、代码组织）的恢复能力，是决定反编译结果工程实用性的核心因素

- 评估各方法在不同优化级别上的可重执行率和可读性
  - ReF Decompile在可重执行率和可读性上均取得了最优结果
  - 基于规则的反编译器在可重执行率上表现最差，在可读性上表现也较差
  - 随着编译优化级别的提升，方法的可重执行率和可读性下降

Model/Metrics	Re-executability Rate (%)					Readability (#)				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
Rule Based Decompiler										
Ghidra	34.76	16.46	15.24	14.02	20.12	2.98	2.41	2.52	2.38	2.57
Refine-Based Method										
GPT-4o	46.95	34.15	28.66	31.10	35.22	2.82	2.35	2.29	2.31	2.44
LLM4Decompile-Ref	74.39	46.95	47.56	42.07	52.74	4.08	3.38	3.34	3.19	3.50
End-to-End Method										
LLM4Decompile-End	69.51	44.51	39.63	38.41	48.02	4.07	3.46	3.40	3.23	3.54
FAE Decompile	67.68	48.78	45.73	42.07	51.07	3.94	3.46	3.40	3.25	3.51
ReF Decompile	**85.37**	**56.10**	**51.83**	**52.43**	**61.43**	**4.13**	**3.60**	**3.54**	**3.49**	**3.69**

- 对比不同参数规模的ReF Decompile和LLM4DecompileRef的可重执行率
  - 同等参数规模下，ReF Decompile的性能优于LLM4DecompileRef
  - 在简单场景下，ReF Decompile更有优势



实验设计　消融实验

- 评估Relabeling与Function Call两种策略对模型性能的影响
  - 两者单独使用均能提升模型性能，且两者结合时表现最佳
  - 重标记策略在O2上并不总是带来可重执行率的提升
  - 对于其他基线大语言模型也是如此，说明这两种策略的普适性

Components		Re-executability Rate (%)					Readability (#)				
🏷	🔧	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
Initialize with LLM4Decompile-End-6.7B (Tan et al., 2024)											
✗	✗	69.51	46.95	50.61	46.34	53.35	3.98	3.41	3.44	3.38	3.55
✓	✗	75.61	50.61	50.00	50.00	56.55	4.01	3.44	3.39	**3.49**	3.58
✗	✓	83.54	**56.10**	51.22	50.61	60.37	4.05	3.51	3.51	3.42	3.62
✓	✓	**85.37**	**56.10**	**51.83**	**52.43**	**61.43**	**4.13**	**3.60**	**3.54**	**3.49**	**3.69**
Initialize with Deepseek-Coder-6.7B-base (Guo et al., 2024)											
✗	✗	59.15	35.98	39.02	37.80	42.99	3.71	3.05	3.16	3.05	3.24
✓	✗	66.46	41.46	38.41	36.59	45.73	3.76	3.17	**3.21**	3.08	3.31
✗	✓	70.73	39.63	39.02	40.24	47.41	3.90	3.17	3.08	3.11	3.31
✓	✓	**79.88**	**45.73**	**43.90**	**42.68**	**53.05**	**3.96**	**3.21**	3.18	**3.19**	**3.38**

- 评估在未微调模型上，Relabeling与Function Call两种策略对模型性能的影响
  - Relabeling策略带来的性能提升表明：只需对输入进行简单的预处理以提升逻辑可读性，也能在无需额外微调的情况下显著提高模型的性能
  - ✓：在提示词中建议使用Function Call策略
  - ✓：强制模型使用Function Call策略及其返回结果

Components		GPT-4o					Qwen2.5-Coder-32B-Instruct				
🏷	🛠	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
✗	✗	30.49	17.68	18.90	18.29	21.34	12.20	12.20	11.59	9.15	11.28
✓	✗	37.80	26.22	24.39	22.56	27.74	15.24	16.46	14.02	12.80	14.63
✗	✓	28.66	15.85	14.63	17.07	19.05	23.17	14.02	15.24	10.37	15.70
✓	✓	31.71	25.61	22.56	25.00	26.22	23.78	16.46	**17.07**	10.37	16.92
✗	✓	35.98	21.95	16.46	18.29	23.17	**25.00**	15.85	16.46	**10.98**	**17.07**
✓	✓	**43.29**	**29.27**	**26.83**	**28.66**	**32.01**	24.39	**17.07**	**16.46**	10.37	**17.07**

算法总结 ReF Decompile

- 算法贡献
  - 针对现有方法丢失控制流结构信息，导致难以准确恢复程序逻辑的问题
    - 通过用直观标签替换特定地址，保留程序控制流信息，并提升程序可读性和逻辑清晰度，从而让模型更容易理解跳转逻辑，得到更好的推理结果
  - 针对现有方法仅依赖于可执行段中的信息，缺失大量变量值相关信息的问题
    - 通过为模型建立函数调用机制，与二进制文件交互，从.rodata段中提取对应数据，从而补全变量信息，提升反编译准确率
  - 算法不足
    - 需依赖外部工具与二进制文件交互，在某些场景下会影响实用性
    - 支持的语言和平台有限

DeGPT: Optimizing Decompiler Output with LLM

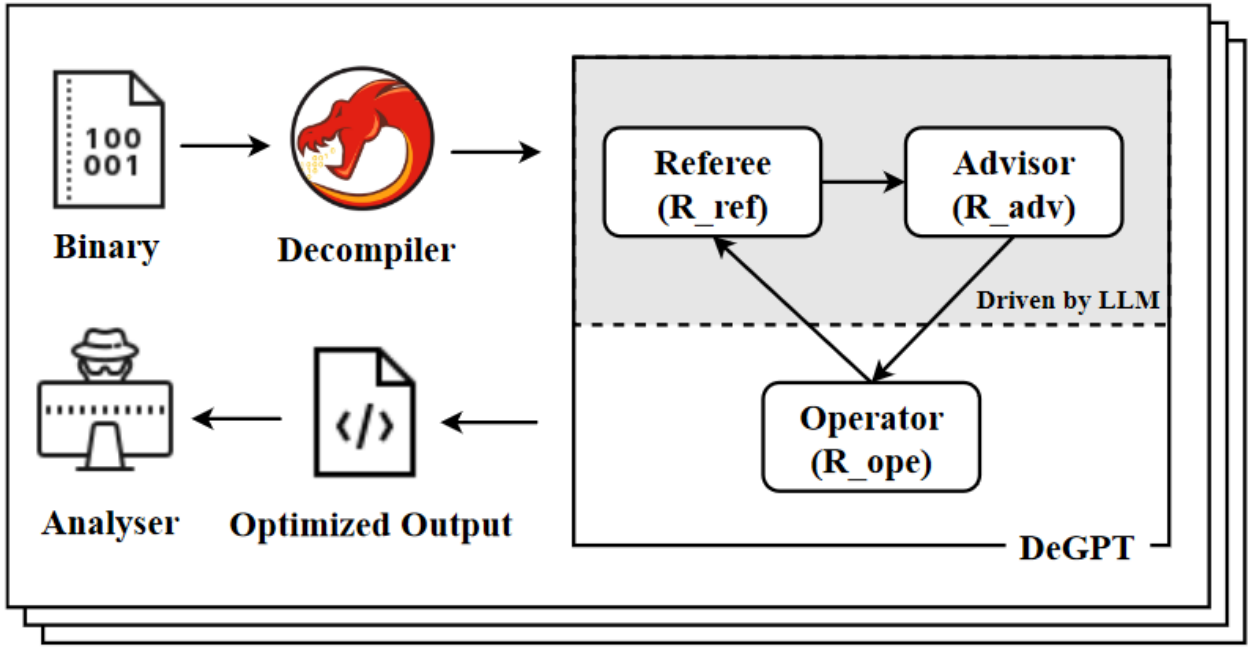
DeGPT TIPO

T	目标	在保持函数语义一致的基础上，优化反编译器输出，增强其可读性与简洁性，促进对二进制程序快速且准确的理解
I	输入	二进制文件*1
P	处理	1. 使用基于规则的反编译器对二进制文件执行反编译 2. 应用三角色机制优化反编译器输出 3. 输出经过优化的反编译结果
O	输出	经过优化的反编译结果*1

P	问题	现有方法仅单步调用LLM，极大地限制了其潜力；LLM输出的不确定性可能会破坏反编译器输出的保真度
C	条件	二进制文件可被反编译器正常反编译
D	难点	如何在保持函数语义一致的基础上增强反编译器输出的可读性和简洁性
L	水平	NDSS 2024 CCF A

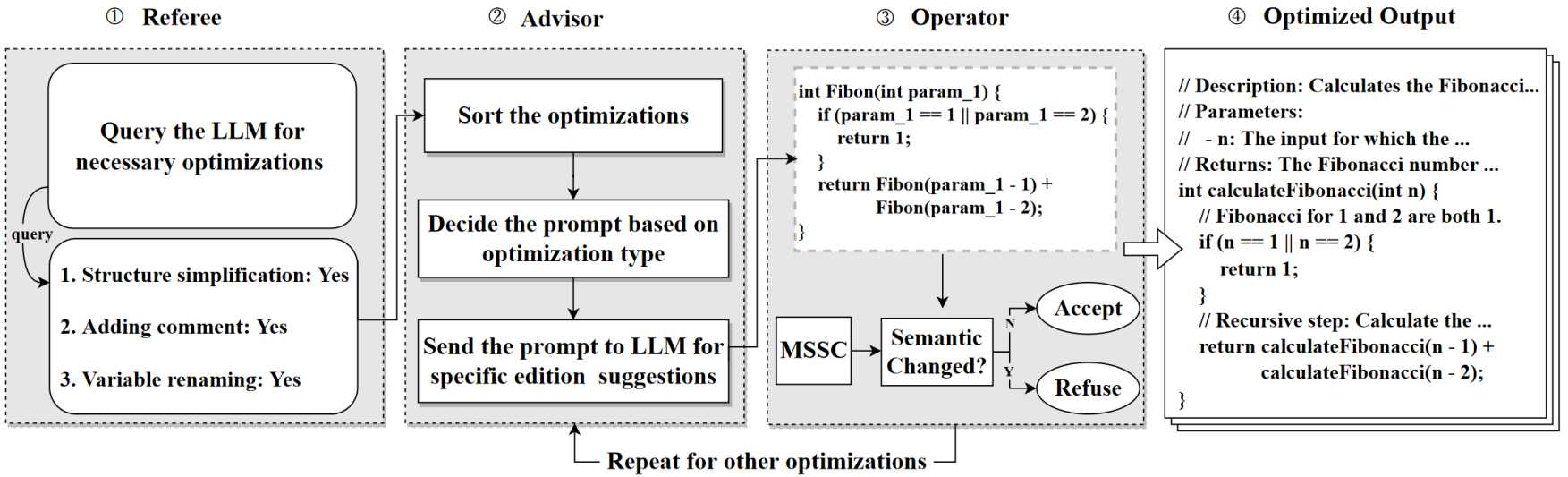
DeGPT 算法原理

- 现有方法仅单步调用LLM，极大限制其潜力；LLM输出的不确定性可能引发语义偏差，破坏与原始反编译器输出的语义一致性
- 算法原理图
  - 使用基于规则的反编译器执行反编译
  - 应用三角色机制（Three-Role Mechanism）优化反编译器输出
    - 将优化任务细粒度拆分，逐步引导LLM，以最大限度地发挥其能力
    - 执行微代码语义计算确保语义一致性
  - 输出经过优化的反编译结果



DeGPT Three-Role Mechanism

- 三角色机制：在保持函数语义不变的前提下，提升代码可读性与简洁性
  - 裁判（R_ref）：提供优化方案。判断哪些优化任务是必要的，避免不必要的开销
    - 优化任务：变量重命名、添加注释、结构简化
  - 顾问（R_adv）：给出实现方案的具体修正措施
    - 优化任务排序；根据优化任务选择专属提示词；发送专属提示词
  - 操作员（R_ope）：通过微代码语义计算检查具体修正措施是否改变原始函数语义，并裁定是否执行优化方案



微代码片段语义计算

- 函数语义一致性定义

$$\begin{cases} F(i) = F'(i) \\ SideEffect(F(i)) = SideEffect(F'(i)) \end{cases}$$

  - 其中，F 为原始反编译器输出，F' 为优化后的输出，$i \in Input(F)$
  - $SideEffect$表示函数调用产生的副作用（side effect）
    - 当函数修改了其局部环境之外的某些状态变量的值，即除了向调用者返回值这一主效应之外，还产生其他可观察的影响时，称其具有副作用
- 微代码片段语义计算（Micro Snippet Semantic Calculation）
  - 原始计算阶段：模拟每条执行路径，记录符号值变化与函数调用信息，为后续语义对比提供基础数据
  - 比较阶段：通过检查函数调用与关键变量值的一致性，判断优化前后代码的语义是否保持不变
  - 属于静态分析方法，与符号执行不同

实验设计

- 完全依赖现有LLM，无需重新预训练或微调
- 测试集
  - 代码库（函数数量）：LeetCode(170)、Coreutils(310)、Mirai(40)、AudioFlux(100)
  - 编译器、编译优化级别、反编译器：GCC 9.4.0、O2、Ghidra 10.2.3
- 基线LLM：GPT-3.5-Turbo
- 对比方法：基于精炼的方法DIRTY
- 评价指标
  - 有意义变量比（Meaningful Variable Ratio, MVR）
    - 优化后输出中有意义的变量占所有变量的比值。评估模型在变量重命名方面的有效性，值越大说明模型性能越好
    - 若优化后的变量名与源代码变量名之间的莱文斯坦距离小于阈值，认为其有意义
      - 对于长度少于5个字符的变量名，仅当名称完全一致时才视为正确；对于更长的变量名，如果其莱文斯坦编辑距离小于名称长度的30%，认为其命名正确

实验设计

- 评价指标
  - 变量重命名比（Optimized Name Ratio, ONR）
    - 表示优化后输出中，被重新分配名称的变量占所有变量的比例。评估模型在变量命名优化方面覆盖的全面性，值越大说明模型性能越好
  - 工作量比（Effort Ratio, ER）
    - 优化后与优化前程序霍尔斯特德复杂度中Effort指标（反映实现或理解程序所需的工作量）的比值。评估模型在结构简化方面的有效性，值越小说明模型性能越好
  - 注释正确比*（Correct Rate of Comments, CR）
    - 优化后输出中的正确注释数占所有注释数的比值。评估模型在注释生成方面的准确性
  - 非平凡注释比*（Non-trivial Rate of Comments, NR）
    - 优化后输出中的非平凡注释数占所有注释数的比值。评估模型生成注释的质量

- 实验结论
  - DeGPT对于剥离二进制文件
    - 有意义变量比、非平凡注释比指标显著下降
    - 工作量比、注释正确比指标基本不变
  - DeGPT在有意义变量比和变量重命名比指标上显著优于DIRTY，说明其能够为更多变量分配更具有实际语义的名称
    - DIRTY效果受限的主要原因：方法采用的变量对齐方式限制了变量重命名的范围，未对齐的变量无法被优化

①/②表示未剥离/剥离二进制文件

Codebase	MVR (%)		ER (%)		CR (%)		NR (%)	
	①	②	①	②	①	②	①	②
LeetCode	27.0	23.0	75.5	76.9	98.7	100	64.8	36.8
Mirai	29.1	17.3	77.5	75.8	99.4	96.8	71.6	36.7
Coreutils	28.0	20.2	72.0	74.2	98.9	100	62.0	40.6
AudioFlux	37.0	36.4	77.6	78.5	99.6	100	53.0	38.4
Average	30.3	24.2	75.6	76.3	99.2	99.2	62.9	38.1

Codebase	DeGPT		DIRTY	
	MVR	ONR	MVR	ONR
LeetCode	27.0%	93.0%	6.2%	78.2%
Mirai	29.1%	94.3%	16.5%	76.3%
Coreutils	28.0%	90.2%	10.7%	75.7%
AudioFlux	37.0%	96.8%	⋆	⋆
Average	30.3%	93.6%	11.1%	76.7%

- 实验结论
  - 应用三角色机制后，相较于单步调用，模型的各类优化频率都有显著提升

①/②表示是/否应用三角色机制

Codebase	Simplify		Comment		Rename		ONR	
	①	②	①	②	①	②	①	②
LeetCode	168	74	161	21	121	55	93.0%	46.0%
Mirai	40	20	37	6	32	16	94.3%	44.5%
Coreutils	301	104	278	109	224	38	90.2%	53.8%
AudioFlux	97	46	81	24	76	35	96.8%	83.6%
Total	606	244	557	160	453	144	92.4%	57.0%

算法总结 DeGPT

- 算法贡献
  - 针对现有方法仅单步调用LLM，极大限制其潜力
    - 应用三角色机制优化反编译器输出，提升代码可读性与简洁性。通过将优化任务细粒度拆分，逐步引导LLM，以最大限度地发挥其能力
  - 针对现有方法LLM输出的不确定性可能引发语义偏差，破坏与原始反编译器输出的语义一致性
    - 通过微代码语义计算确保优化前后的函数语义一致性
- 算法不足
  - 算法仅依赖于可执行段中的信息，未考虑函数调用上下文
  - 目前的微代码片段语义计算方法仅能处理简单函数，函数复杂度的增加将导致执行路径数量呈指数级增长

特点总结与未来展望

特点总结与未来展望

- 特点总结
  - ReF Decompile
    - 端到端方法
    - 重点关注如何提升反编译代码的准确性
    - 结合二进制程序分析知识，补全了现有方法缺失的控制流结构和变量信息，有效提升了反编译代码的准确性和可读性
  - DeGPT
    - 基于精炼的方法
    - 重点关注如何提升反编译代码的可读性和简洁性
    - 结合LLM领域相关知识，进一步发挥LLM在该任务上的潜力，提升反编译代码的可读性和简洁性；结合源代码程序分析知识，确保优化前后代码的语义一致性
- 未来发展
  - 结合更多的二进制与源代码程序分析技术与领域相关知识针对性地提升方法的性能

参考文献

[1] Feng Y, Li B, Shi X, et al. ReF Decompile: Relabeling and Function Call Enhanced Decompile[J]. arXiv preprint arXiv:2502.12221, 2025.

[2] Hu P, Liang R, Chen K. DeGPT: Optimizing Decompiler Output with LLM[C]. Proceedings of the 2024 Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2024: 267622140.

道德经

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！