软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(6):2327−2345 [doi: 10.13328/j.cnki.jos.007616] [CSTR: 32375.14.jos.007616]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

BinDec: 面向 RISC-V 的 LLM 与符号执行协同反编译方法*

李玉璋[1], 张 熙[1], 徐 涛[2]

[1](北京邮电大学 网络空间安全学院, 北京 100876)

[2](清华大学 计算机科学与技术系, 北京 100084)

通信作者: 徐涛, E-mail: xtao@tsinghua.edu.cn

摘　要: 反编译是软件逆向工程中的基础技术, 其目标是从面向硬件的二进制代码中恢复出高级语言代码, 以支持人工阅读、分析或重工程任务. 尽管该技术已得到广泛研究, 但传统基于规则的反编译器所生成的反编译代码往往可读性较差, 且难以复用. 此外, 由于传统反编译器的开发周期较长, 其对 RISC-V 等新兴指令集架构的支持通常较为滞后. 在当前大语言模型 (large language model, LLM) 技术广泛应用于自动化软件工程任务并取得显著成效的背景下, 面向 RISC-V 架构的反编译需求, 提出了一种 LLM 与符号执行协同的反编译方法 BinDec. 该方法通过 LLM 生成与符号执行验证的交替迭代, 充分利用 LLM 的代码理解与生成能力, 以产生更易于理解与重用的反编译代码; 同时借助符号执行的代码分析与验证能力, 确保生成结果的可靠性. 通过一系列实验对 BinDec 的有效性进行了评估, 实验结果表明, 该方法在达到与传统反编译器相近的语义准确性的同时显著提升了代码的可读性.

关键词: 反编译; 大语言模型; 符号执行

中图法分类号: TP311

中文引用格式: 李玉璋, 张熙, 徐涛. BinDec: 面向RISC-V的LLM与符号执行协同反编译方法. 软件学报, 2026, 37(6): 2327–2345. http://www.jos.org.cn/1000-9825/7616.htm

英文引用格式: Li YZ, Zhang X, Xu T. BinDec: LLM and Symbolic Execution Collaborative Decompilation Method for RISC-V. Ruan Jian Xue Bao/Journal of Software, 2026, 37(6): 2327–2345 (in Chinese). http://www.jos.org.cn/1000-9825/7616.htm

BinDec: LLM and Symbolic Execution Collaborative Decompilation Method for RISC-V

LI Yu-Zhang[1], ZHANG Xi[1], XU Tao[2]

[1](School of Cyberspace Security, Beijing University of Posts and Telecommunications, Beijing 100876, China)

[2](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

Abstract: Decompilation serves as a fundamental technique in software reverse engineering, aiming to recover high-level source code from hardware-oriented binary programs to support human understanding, analysis, and re-engineering tasks. Although this technique has been extensively studied, traditional rule-based decompilers often generate decompiled code with poor readability and limited reusability. Moreover, due to long development cycles, support for emerging instruction set architectures such as RISC-V is typically delayed in conventional decompilers. With the widespread adoption of large language models (LLMs) in automated software engineering tasks and their demonstrated effectiveness, this study proposes BinDec, a RISC-V binary decompilation approach that synergistically integrates LLM and symbolic execution. The proposed method alternates between LLM-based code generation and symbolic execution-based verification, fully exploiting the code understanding and generation capabilities of LLM to produce decompiled code that is more readable and reusable, while leveraging the analysis and verification capabilities of symbolic execution to ensure semantic correctness and reliability. The effectiveness of the proposed method is evaluated through a series of experiments. Experimental results demonstrate that BinDec achieves semantic accuracy comparable to that of traditional decompilers, while significantly improving the readability of the generated decompiled code.

* 本文由“RISC-V 与人工智能系统软件前沿进展”专题特约编辑武延军研究员、谢涛教授、侯锐研究员、宋威研究员、邢明杰高级工程师推荐.
收稿时间: 2025-09-07; 修改时间: 2025-10-20, 2025-12-08; 采用时间: 2025-12-17; jos 在线出版时间: 2025-12-26
CNKI 网络首发时间: 2026-03-12

Key words: decompilation; large language model (LLM); symbolic execution

反编译技术是计算机与信息安全领域的一项关键基础技术, 广泛应用于软件漏洞与缺陷分析、逆向工程以及软件重用与维护等任务中[1,2]. 作为编译过程的逆操作, 反编译的核心任务在于恢复源代码经编译优化生成目标代码过程中所丢失的高级语义信息, 例如函数与变量名称、控制逻辑及结构体定义等[3−7]. 这些信息不仅是人工编写代码的主要特征, 也是保障代码可读性的关键要素. 传统反编译方法为完成从二进制代码到高级语言的恢复任务, 通常依赖工程经验定义大量启发式规则, 并借助高级语言代码模板对二进制代码片段进行匹配. 然而, 由于启发式规则仅能针对特定模式进行有限恢复, 其输出结果往往难以达到接近人工编写代码的可读性水平, 尤其表现为变量命名缺乏语义信息、控制流结构混乱等问题, 导致反编译结果通常需由专业人员深入分析后方可使用. 此外, 传统反编译器的可用性高度依赖于针对不同指令集架构的适配工作, 而 RISC-V 作为一种新兴指令集架构, 目前仅有少数反编译器能够提供较为成熟的支持, 例如 IDA Pro 和 Ghidra.

近年来, 随着深度学习技术的迅速发展, 尤其是自然语言处理任务中取得的显著成果, 部分研究开始尝试将相关技术应用于反编译任务中[8,9]. 基于学习的反编译框架将二进制代码到源代码的转换视为一种神经机器翻译 (neural machine translation, NMT) 任务, 利用大规模源代码与二进制代码的配对样本训练端到端的反编译模型. 神经反编译器通过从代码数据集中学习人工编写代码的潜在特征, 生成的代码在可读性方面显著优于传统反编译方法. 随着大语言模型 (large language model, LLM) 的爆发式发展, 基于 LLM 生成的反编译代码的可读性则进一步提升. 因此, 基于学习的技术被认为有望改善反编译过程中高级语义信息恢复的难题. 然而, 以 LLM 为代表的深度学习方法本质上是基于概率的黑盒模型, 存在输出不确定性与可解释性差等问题. 例如, LLM 常被报告存在幻觉现象, 即生成内容与事实不符, 这为其在反编译任务中的应用带来严峻挑战. 由于反编译的核心目标是准确分析二进制代码的功能, 若假定 LLM 生成的结果不可靠, 而用户又缺乏其他途径验证其正确性, 将严重限制该类方法在实际场景中的可用性.

为解决当前面向 RISC-V 架构的反编译工具选择有限以及基于学习的反编译方法存在结果不确定性的问题, 本文提出了一种基于 LLM 的 RISC-V 二进制反编译方法 BinDec. 该方法融合 LLM 与符号执行技术, 实现对 RISC-V 二进制代码到 C 语言代码的反编译. 具体而言, BinDec 方法首先利用 LLM 将二进制代码提升为 LLVM 中间代码 (LLVM intermediate representation, LLVM IR, 以下简称为 IR); 随后, 通过符号执行生成测试用例, 并分别在提升得到的 IR 与原始二进制程序上执行, 以验证两者行为的一致性; 接着, 再次调用 LLM 将 IR 反编译为 C 代码; 最后, 将所得 C 代码重新编译为 IR, 通过构建反编译前后两个版本 IR 的融合符号模型, 并借助 SMT (satisfiability modulo theories)[10]求解器判断其等价性约束的可满足性, 从而验证反编译 C 代码与提升得到的 IR 之间的语义一致性. 在验证过程中, 若 LLM 输出无法编译或未通过等价性检查, BinDec 方法将迭代式地请求 LLM 进行修复或重新生成. BinDec 方法的关键在于将反编译任务划分为二进制代码提升与中间代码反编译两个阶段, 从而为符号执行这一作用于中间代码层的验证技术提供应用机会. 基于这一架构, BinDec 方法能够在无需额外辅助信息的条件下, 独立完成 RISC-V 二进制代码的反编译与结果验证, 确保向用户提供可验证的高质量反编译结果. 实验结果表明, BinDec 方法在语义正确性方面达到了与传统反编译器 Ghidra 相当的水平, 同时其生成的代码在文本可读性方面显著优于后者. 具体而言, 以文本相似性作为衡量指标时, BinDec 方法生成代码的可读性约为 Ghidra 的两倍. 综上所述, 本文的贡献总结如下.

(1) 提出一种面向 RISC-V 架构的反编译方法 BinDec, 实现对 RISC-V 二进制代码的反编译, 可作为该架构下反编译任务的一种可行解决方案.

(2) 提出将 LLM 与符号执行相结合的协同方法, 在实现充分利用 LLM 的代码理解与生成能力的基础上, 借助符号执行的形式化能力为 LLM 输出的代码提供可靠性保证.

(3) 实现 BinDec 方法并开展详尽的实验评估, 验证其有效性.

本文第 1 节介绍反编译的研究背景以及相关的研究工作. 第 2 节介绍本文提出的方法. 第 3 节介绍本文的实验设计, 包括研究问题、基准方法、实验数据、评估指标和实现细节. 第 4 节报告实验结果和针对每个研究问题

的分析. 第 5 节讨论本文方法的局限性和实验效度威胁. 第 6 节对本文进行总结并展望未来的研究.

1 研究背景和相关工作

本节介绍反编译的研究背景以及相关的研究工作. 首先说明传统反编译器的背景知识, 然后介绍与本文相关的神经反编译相关工作.

1.1 研究背景

反编译器的核心功能是将低级语言代码恢复为高级语言代码. 反编译器通常处理两类低级语言: 一类是面向特定硬件指令集架构的二进制代码; 另一类是基于虚拟机实现的程序语言所对应的字节码, 例如 Java 字节码. 字节码的反编译通常属于语言支持的一部分, 实现相对容易; 而面向硬件的二进制代码反编译则是一个更为广泛且复杂的问题. 本文所研究的反编译问题限定于针对二进制代码的情形. 与反编译密切相关的概念是反汇编, 后者指通过解析二进制代码中的指令序列, 将其转换为等价的汇编代码. 反汇编器通常作为编译工具链的组成部分, 例如 GCC 编译器工具链中包含针对不同硬件架构的 objdump 工具, 可用于完成反汇编操作.

反编译器的基本工作流程是: 首先对二进制代码进行反汇编得到汇编代码, 随后使用针对特定指令集架构的提升器将得到的汇编代码提升为该反编译器内部所使用的中间表示形式; 在此基础上, 进一步开展控制流与数据流分析, 以恢复高级语言代码的结构与语义特征; 最终生成目标高级语言形式的反编译代码. 其总体过程如图 1.

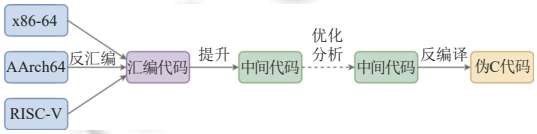


图 1 反编译器总体架构

中间代码在反编译器中起到关键作用, 作为一种代码抽象形式, 它为多个代码分析模块提供统一的输入与输出接口. 表 1 展示了主流反编译产品或项目中所采用的中间表示. 可以看出, 大多数反编译器都为其自身定义了中间代码的专用语言. 一个例外是 RetDec, 其采用了 LLVM 工具链中广泛使用的 LLVM IR 作为其中间表示. LLVM IR 是 LLVM 编译器基础设施中原生支持的中间代码形式, 使用它作为反编译器的中间代码有助于复用或集成 LLVM 工具链中的现有处理能力. 本文所提出的方法同样采用 LLVM IR 作为中间代码.

表 1 主流反编译产品或项目

名称	性质	中间代码	目标代码	RISC-V支持	网址
IDA Pro	商业	microcode	C	支持	https://hex-rays.com/ida-pro
JEB Decompiler	商业	未知	C	部分支持	https://www.pnfsoftware.com/jeb
Binary Ninja	商业	BNIL	C	支持	https://binary.ninja
Ghidra	开源	P-code	C	支持	https://github.com/NationalSecurityAgency/ghidra
angr	开源	VEX IR	C	不支持	https://angr.io
RetDec	开源	LLVM IR	C	不支持	https://github.com/avast/retdec

当前, 基于上述技术路线的反编译技术虽已成熟并催生了众多实用工具 (如表 1 所示), 但其在生成代码的可读性和开发成本两方面均存在固有局限. 首先, 在可读性方面, 由于编译过程中源代码中的诸多可读性要素 (如变量名称、结构体定义与控制流结构) 被大量剥离, 而传统反编译方法主要依赖语法与结构分析, 难以有效重建在编译阶段丢失的高级语义信息. 因此, 基于传统方法生成的反编译代码虽然能够保持功能正确, 但其可读性较差, 例如变量常被命名为“var1”等无意义标识符, 缺乏语义表达力. 这类输出通常被称为“伪代码”, 显著降低了人工分析的效率. 受此影响, 反编译结果难以直接复用, 而在面对因源码遗失而需进行软件系统改造的任务时, 代码重用具有重要的经济价值. 其次, 在开发成本方面, 现有反编译器普遍依赖针对特定指令集人工编写的规则库, 导致其开发周期长、实现成本高, 且跨架构适应能力不足. 以 Avast 公司为例, 其历经了 7 年研发并于 2017 年开源的反编

译工具 RetDec, 仅支持 x86 与 ARM 等少数指令集架构. 该项目自 2022 年开始因资源不足进入有限维护状态而停止功能更新, 进一步反映出此类工具在可持续开发方面的挑战. 表 1 统计了当前主流反编译器对 RISC-V 架构的支持情况, 结果显示仅有少数工具能够有效处理 RISC-V 目标代码. 此外, RISC-V 指令集的可扩展特性导致其生态中出现多种扩展指令集, 造成一定程度的架构碎片化. 针对这一问题, 现有反编译工具普遍缺乏原生适配能力, 必须通过额外开发才能支持包含扩展指令的代码, 进一步加重了工程负担与维护难度.

近年来, 随着深度神经网络模型在代码分析与处理任务中的广泛应用[11−18], 基于神经网络的反编译技术逐渐受到关注. 这类方法旨在利用深度学习模型强大的表示与推理能力, 以数据驱动的方式替代传统反编译工具中依赖人工设计的复杂规则. 通过在大规模代码数据上训练端到端的反编译模型, 或对已有的通用代码模型进行反编译任务适配, 神经网络辅助的反编译系统能够生成结构更清晰、可读性更高的代码. 特别是随着 LLM 技术的显著进展, 其从海量代码语料中获得的丰富语义知识、出色的上下文理解能力以及优秀的泛化性能, 为解决上述传统反编译器面临的两大核心挑战提供了新的思路. 具体来说, LLM 在代码理解与生成方面展现出强大潜力, 能够生成语义更贴合上下文的目标代码. 例如, 在变量命名时能够恢复具有实际意义的标识符名称, 而非简单的“var1”类占位符. 另一方面, 通过对 LLM 进行有针对性的微调, 可以进一步激发其跨指令集架构的泛化能力, 从而为支持新硬件平台的反编译任务提供一种灵活且可扩展的解决方案.

然而, 需要指出的是, 神经网络方法本身存在一定的固有不确定性, 并可能产生幻觉现象, 导致生成结果中出现看似合理但不正确的代码片段. 因此, 如何保障神经反编译系统输出结果的可靠性与正确性, 仍是当前研究中亟待解决的关键问题. 针对传统反编译器的可靠性验证, Zou 等人[19]提出了测试框架 D-Helix, 其核心方法是利用符号执行技术检查反编译中间代码与高级语言代码之间的一致性, 从而发掘反编译器中存在的缺陷. 然而, D-Helix 依赖预定义的规则对编译错误进行修复, 这类规则往往覆盖有限, 且可能引入语义失真, 从而限制了该框架的实用性. 本文探索了基于 LLM 修复编译错误, 并结合符号执行对修复后代码进行语义等价性验证的协同方法. 该方法将 D-Helix 的验证思路扩展至神经反编译场景, 从而为神经反编译的可靠性验证提供可行的技术支撑.

1.2 相关工作

本节介绍神经反编译技术的相关研究进展. 基于深度神经网络的反编译技术致力于将自然语言处理领域的深度模型成功经验迁移至代码反编译任务中, 该领域的研究发展可划分为 3 个阶段.

第 1 阶段以循环神经网络 (recurrent neural network, RNN) 为主要模型基础. 在该阶段中, 研究者将反编译视为一种特殊的神经机器翻译任务, 并利用代码的领域知识设计专门的特征输入方法. 与常规机器翻译任务缺乏双语语料库不同, 针对代码的反编译任务可利用编译器自动生成大量配对的高级语言与低级语言样本. Katz 等人[8]将基于 RNN 的序列到序列模型应用于从低级代码到高级代码的翻译任务. 随后, Fu 等人[9]指出原始的序列到序列模型因缺乏对底层代码领域知识的建模能力而不适用于反编译任务, 并提出了多模型协同的神经反编译器框架 Coda. 该框架使用不同的神经网络分别承担反编译代码生成与错误预测子任务, 进而引入自动纠错机制以提升生成代码的可靠性. 这一阶段的研究初步确立了将神经网络应用于反编译任务的基本范式, 即借助编译器自动构建大规模配对样本, 并基于此训练语言模型. 然而, 受限于 RNN 架构难以有效建模程序代码中的长距离依赖和复杂语法结构, 这导致反编译结果的正确性和可读性较差, 仅在实验设定下的简单代码片段上取得一定效果.

第 2 阶段的发展与 Transformer 模型[20]的广泛成功密切相关. 研究者尝试将更大规模的语言模型应用于反编译任务. Hosseini 等人[21]提出了基于 Transformer 的反编译模型 BTC, 其特点是将源代码视为纯文本输入, 从而避免在特征构建过程中依赖代码领域知识, 降低了模型跨编程语言迁移的复杂性. 在 Go、Fortran、OCaml 和 C 等多种语言上进行的实验表明, BTC 达到了与已有方法相当的性能. Al-Kaswan 等人[22]构建了一个包含 21.4 万个样本的反编译数据集 CAPYBARA, 并在该数据集上对预训练语言模型 CodeT5[23]进行微调, 得到反编译模型 BinT5. 该模型在反编译任务中取得了 58.82% 的 BLEU-4 分数. 此后, Armengol-Estapé等人[24]提出反编译模型 SLaDe, 其特点是为源代码设计了专用分词器, 并利用 PsycheC[25]工具为模型输出的反编译代码补全类型信息. 实验结果表明, SLaDe 生成代码的准确率优于传统反编译器 Ghidra. Jiang 等人[26]提出基于大语言模型预训练的反编译模型 Nova, 该模型针对编译优化问题设计了优化生成与优化级别预测两个预训练任务, 从而在优化级别较高的汇编代

码反编译中表现出色. 总体来看, 该阶段研究的发展主要受益于 Transformer 模型架构的广泛应用以及大规模数据预训练范式的成功实践, 使得反编译任务在性能上取得显著提升, 并展现出一定的泛化能力. 然而, 当前研究在走向实用化的过程中仍面临若干关键限制. 首先, 模型所能处理的上下文长度通常较为有限 (例如仅支持 512 个 token), 导致其难以有效处理长代码片段. 其次, 现有基于大规模预训练的反编译模型通常仅支持将单一指令集架构的二进制代码转换为某一种特定的高级编程语言, 这种一对一的映射方式在实际应用中的适应性与性价比均存在明显局限.

第 3 阶段的研究主要集中于对 LLM 进行提示工程以实现反编译, 重点是利用 LLM 对传统反编译器生成的代码进行优化. Xu 等人[27]提出了结合 LLM 与程序分析技术的反编译代码优化框架 GenNm. 该框架通过程序分析提取反编译代码片段, 构造提示词请求 LLM 生成变量命名建议, 并再次借助程序分析完成全局变量替换, 从而提升代码可读性. Wong 等人[28]提出了基于 GPT-3.5/4 的反编译框架 DecLLM, 其首先使用 IDA Pro 将二进制代码反编译为伪 C 代码, 随后尝试编译和执行该代码, 针对编译或执行过程中出现的错误, 构造提示词请求大语言模型对代码进行修复. Hu 等人[29]提出了基于三角色机制的反编译代码优化框架 DeGPT, 将对 LLM 的查询划分为优化建议与优化执行两个阶段, 并进一步通过代码模拟执行实现语义一致性检查, 以评估优化后代码的质量. 相较于前两个研究阶段, 本阶段工作主要立足于 LLM 所具备的强大能力, 通过零样本学习的方式将其直接应用于反编译任务中. 由于 LLM 能够生成质量较高的代码, 研究重点已不再局限于评估生成代码与参考代码之间的相似性, 而是进一步转向关注其实际可用性, 例如生成代码是否具备可编译性与可执行性等实用属性.

回顾神经反编译研究的 3 个阶段, 可以发现尽管现有研究已构建了将神经网络模型应用于反编译任务的基本范式, 但仍存在若干尚未解决的问题. 首先, 当前将 LLM 应用于反编译任务时, 主要仍以辅助形式出现. 例如, DecLLM 与 DeGPT 均利用 LLM 对现有反编译器的输出进行修复与完善. 然而, 与早期基于小规模语言模型的反编译方法已取得的成效相比, 我们认为此类范式未能充分发挥 LLM 在代码理解与生成方面的潜力, 并可能继承现有反编译器固有的局限性, 例如语义还原不准确以及对新硬件架构支持滞后等问题. 其次, 当前研究普遍缺乏对生成的反编译代码进行可靠验证的机制. 尽管部分研究已开始关注验证问题, 例如 SLaDe[24]基于类型系统对反编译代码进行了修正与完善, DeGPT[29]提出了基于代码路径模拟的评估方法, 但尚未有研究系统性地提出保障反编译代码可靠性的综合措施. 最后, 现有研究对反编译代码的粒度关注不足, 现实世界中程序规模差异显著, 若忽略这种差异, 可能导致神经反编译方法在处理大规模代码时超出模型上下文长度的限制, 进而导致方法失效.

针对上述问题, 本研究提出了 BinDec 方法. 首先, 我们探索了直接将二进制代码作为 LLM 输入的可行性, 以避免对现有反编译器的依赖, 从而使该方法能够适用于 RISC-V 等新兴架构. 其次, 我们将符号执行技术引入反编译流程, 对所有生成的反编译代码实施严格的形式化验证, 以保障其正确性. 最后, 我们聚焦于函数粒度的二进制代码反编译, 并恢复函数间的调用关系, 以支持程序级别反编译代码的完整重建.

2 基于大语言模型与符号执行协同的反编译方法

本节介绍本文所提出的基于大语言模型的两阶段反编译方法 BinDec. 该方法结合了大语言模型的文本理解与生成能力以及基于符号执行的代码分析技术, 通过大语言模型生成与符号执行判别之间的多轮迭代, 首先将二进制代码提升为中间代码, 继而将其进一步反编译为 C 代码. 本节将从总体方法、二进制代码提升、中间代码反编译、LLM 提示工程和函数级符号执行这 5 个方面对该方法进行介绍.

2.1 总体方法

BinDec 方法的关键是结合 LLM 与符号执行技术, 以协同迭代的方式实现二进制代码的反编译 (如图 2 所示). 其具体流程如下.

(a) 二进制代码信息提取: 利用现有二进制代码分析工具, 以函数为单位从二进制文件中提取代码信息, 包括入口地址、指令序列、调用关系及数据依赖等 (图 2 步骤 (1)).

(b) 二进制代码提升: 根据函数的指令序列构造提示词, 请求 LLM 生成语义一致的 LLVM IR (以下简称 IR)

代码 (步骤 (2)). 对生成的代码进行编译检查 (步骤 (3)), 若发现错误, 则重新构造提示词请求 LLM 修复 (步骤 (4)), 直至获得语法正确的 IR 代码.

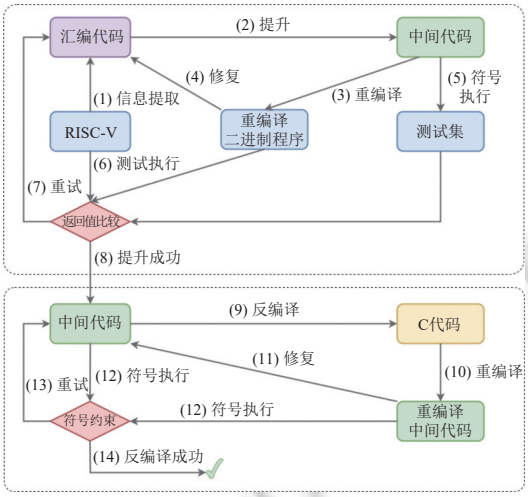


图 2 BinDec 方法总体架构

(c) 中间代码检查: 对提升得到的 IR 代码进行符号执行 (步骤 (5)), 生成一组测试用例. 在原始二进制代码和提升后的 IR 上分别运行这些测试用例 (步骤 (6)), 并对比运行结果, 以验证中间代码与原始二进制代码的一致性. 若结果不一致, 则返回上一阶段要求 LLM 重新生成 (步骤 (7)).

(d) 中间代码反编译: 基于已提升的 IR 代码构造提示词, 请求 LLM 生成相应的 C 代码 (步骤 (9)). 对生成的 C 代码进行编译检查 (步骤 (10)), 如存在错误, 则重新构造提示词请求 LLM 修复 (步骤 (11)), 直至获得语法正确的 C 代码.

(e) 源代码检查: 将反编译得到的 C 代码重新编译为 IR 代码 (步骤 (10)), 并将其与前一阶段获得的 IR 进行合并. 随后通过符号执行生成等价性约束 (步骤 (12)), 并借助 SMT 求解器进行验证. 若约束求解失败, 则返回上一阶段要求 LLM 重新生成 C 代码 (步骤 (13)).

通过以上流程, BinDec 方法旨在有效结合 LLM 与符号执行, 实现具有可靠性保证的反编译. 具体来说, BinDec 方法以反汇编器分析为基础, 通过自动化的编译反馈循环和两阶段符号执行验证, 提供了可编译性与高置信度的功能等价性保障. 当输入理想时, 其能输出可编译且功能等效的高质量代码; 当处理困难代码时, 则通过明确的失败报告实现优雅降级, 从而确保结果的可靠性与可解释性.

2.2 二进制代码提升

本节阐述将 RISC-V 二进制代码提升至 IR 代码的过程, 其关键是利用 LLM 生成功能等价的 IR 代码. 考虑到二进制代码的整体规模可能较大, 而实际编程中单个函数的规模通常是有限的, 因此本方法选择在函数粒度上对二进制代码进行处理.

为准确描述二进制函数的特征, BinDec 借助反汇编工具 (如 riscv64-linux-gnu-objdump 或 Ghidra) 提取目标函数的入口地址、指令序列、函数调用关系以及数据依赖信息. 入口地址和包含偏移量信息的指令序列完整刻画了二进制函数的执行逻辑, 而函数调用关系与全局数据依赖信息则为后续基于多个独立反编译函数重建完整程序提供了基础. 这些信息被组织为结构化提示词, 输入至 LLM 中, 请求生成对应的 IR 代码. 提示词中包含示例指令与相应 IR 片段的映射关系, 以引导模型生成符合语法规范的代码. 对于 LLM 输出的 IR 代码, BinDec 使用 LLVM IR 编译器 (llc) 进行语法合法性验证. 若发现错误, BinDec 将捕获 llc 输出的错误信息, 构造修复性提示词反馈至 LLM, 要求其根据错误类型 (如类型不匹配、指令不支持或语法错误) 进行修正. 该过程迭代执行, 直至获得语法正确的 IR 代码或达到预设的最大尝试次数.

由于 LLM 本身存在固有的不确定性, 即使其输出的 IR 代码可通过编译, 仍无法保证其与原始二进制代码在功能上完全一致. 为确保语义一致性, BinDec 采用基于符号执行的单元测试方法对生成的 IR 进行验证. 首先, 为提升后的 IR 函数生成一个包含 main 函数的可执行程序 (称为产品程序, 如图 3 所示). 该 main 函数包括对目标函数参数的符号化声明、函数调用以及运行结果输出等部分, 其生成基于对目标函数的签名分析和预设的产品程序模板. 需要注意的是, 目标函数可能调用外部函数. 由于 BinDec 以函数粒度运行, 一种直接的处理方式是按照调用顺序依次处理各函数, 以确保外部函数已被提前反编译. 然而, 这种方式可能导致外层函数涉及复杂的调用关系, 显著增加符号执行的复杂度. 为此, 本文采用简化外部函数建模的策略, 参考已有研究[19]将外部函数定义为直接返回所有参数最低字节之和. 与实际函数定义相比, 该方法大幅提升了符号执行的效率; 同时, 由于所有参数均被使用, 避免了因路径中断而影响符号执行的准确性. 随后, 对产品程序进行符号执行, 并根据搜索到的路径生成多个测试用例. 接下来, 将产品程序编译为 RISC-V 目标二进制程序, 并通过二进制注入技术将其中的 main 函数及所有模拟的外部函数定义注入原始二进制程序中, 从而实现对原始二进制代码中对应函数的独立执行. 最后, 分别运行产品程序与注入后的原始二进制程序, 通过对比不同测试用例下的函数返回值, 判断提升后的 IR 代码与原始二进制代码的语义是否一致. 若出现返回值不一致或执行崩溃, 则判定为语义不一致, 此时丢弃当前提升结果, 重新请求 LLM 生成 IR 代码并重复上述验证流程, 直至获得在测试集上完全等价的结果或达到最大尝试次数.

```
3  #include <stdio.h>
4  #include <klee/klee.h>
5
6  __attribute__((noinline)) int add(int arg0, int arg1) {
7    return (int)((unsigned char)arg0 + (unsigned char)arg1);
8  }
9
10 __attribute__((noinline)) int mul(int arg0, int arg1) {
11   return (int)((unsigned char)arg0 + (unsigned char)arg1);
12 }
13
14 extern int f(int arg0, int arg1);
```

```
16 int main(int argc, char *argv[]) {
17   int arg0;
18   klee_make_symbolic(&arg0, sizeof(arg0), "arg0");
19   int arg1;
20   klee_make_symbolic(&arg1, sizeof(arg1), "arg1");
21
22   int _func_rv = f(arg0, arg1);
23   (void)_func_rv;
24
25   printf("{\"output\": \"0x");
26   for (int i = 0; i < sizeof(_func_rv); ++i) {
27     printf("%02x", ((unsigned char *)&_func_rv)[i]);
28   }
29   printf("\"}");
30   return 0;
31 }
```

图 3　用于单元测试的产品程序示例

2.3　中间代码反编译

在基于第 2.1 节方法获得可编译且语义与原始二进制代码一致的 IR 代码后, BinDec 进一步将其反编译为 C 代码. 本节将介绍从 IR 代码反编译为 C 代码的具体过程, 主要包括 LLM 查询、C 语言代码重编译以及语义验证这 3 个步骤.

本阶段仍以 LLM 为核心, 利用其代码生成能力实现从中间代码到高级语言表示的转换. 与二进制代码提升阶段不同, 本阶段的输入为 IR 代码. 由于 IR 代码具有平台无关性且具备一定的可读性, 因此相比二进制代码更易于 LLM 理解. 为进一步降低 LLM 理解 IR 代码的难度, BinDec 首先对输入的 IR 代码进行规范化处理. 具体做法包括将变量和标签全部匿名化 (即统一表示为类似“%1”的无命名形式), 并移除所有非必要的函数和变量属性信息 (例如用于指导代码优化的属性). 在常规编译流程中, 这类命名和属性信息通常由编译器前端生成, 以维持可读性或指导后续优化步骤. 然而, 对于通过 LLM 从二进制代码提升得到的 IR 代码, 此类信息往往不准确甚至无意义. 移除这些信息有助于降低 LLM 的理解负担, 并减少错误信息带来的干扰. 随后, BinDec 基于规范化后的 IR 代码片段构造提示词, 要求 LLM 生成具备可读性的 C 代码. 对于 LLM 输出的 C 代码, BinDec 采用与二进制代码提升阶段类似的方法进行可编译性修复, 即通过 C 编译器对代码进行编译, 并根据编译器报错信息构造提示词引导 LLM 迭代修复代码, 直至生成语法正确、可编译的 C 代码或达到预设的最大尝试次数.

为保障反编译所得 C 代码与输入 IR 代码之间的语义一致性, BinDec 在本阶段继续采用符号执行技术进行等价性检验. 但与前一阶段不同的是, 本阶段不再依赖由符号执行生成的具体测试用例进行具体执行, 而是完全在符号执行框架下完成判别. 该方法可减少等价性检查的步骤, 从而提升验证效率. 这一差异源于符号执行技术的适用

场景: 符号执行通常适用于中间代码层面的分析. 在二进制代码提升阶段, 因需对比提升后的 IR 代码与原始二进制代码, 必须在二进制层面进行等价性检查; 而在本阶段, 由于比较双方均为中间代码, 因此可以借助全面的符号化验证实现更为严格的等价性检查. 具体而言, BinDec 首先将反编译得到的 C 代码重新编译为 IR 代码, 随后借鉴差分符号执行[30]的思路, 在同一个符号化程序中对两个版本的 IR 代码进行等价性约束求解. 该过程主要包括以下步骤: 首先, 基于两个 IR 代码版本构建一个统一的产品程序 (如图 4 所示). 该程序结构与此前阶段类似, 包含外部函数调用定义和 main 函数两部分, 区别在于本阶段将同一组符号化函数参数同时传入两个目标函数中, 并插入对二者返回值的等价性断言. 接着, 将该产品程序编译为 IR 代码形式, 并将待比较的两个 IR 模块整合至该主程序中以进行符号执行. 最后, 通过 SMT 求解器在符号模型上对目标函数返回值的等价性断言进行求解, 据此判断两个版本是否语义一致. 若断言失败, 则认为语义不一致, 此时丢弃当前反编译结果并重新调用 LLM 生成新的 C 代码. 该过程循环执行, 直至获得通过语义验证的 C 代码或达到最大尝试次数.

```c
#include <stdio.h>
#include <klee/klee.h>

__attribute__((noinline)) int add(int arg0, int arg1) {
  return (int)((unsigned char)arg0 + (unsigned char)arg1);
}

__attribute__((noinline)) int mul(int arg0, int arg1) {
  return (int)((unsigned char)arg0 + (unsigned char)arg1);
}

extern int f_a(int arg0, int arg1);
extern int f_b(int arg0, int arg1);
```

```c
int main(int argc, char *argv[]) {
  int arg0;
  klee_make_symbolic(&arg0, sizeof(arg0), "arg0");
  int arg1;
  klee_make_symbolic(&arg1, sizeof(arg1), "arg1");

  int _func_rv_a = f_a(arg0, arg1);
  int _func_rv_b = f_b(arg0, arg1);
  for (int i = 0; i < sizeof(_func_rv_a); ++i) {
    unsigned char _func_rv_a_i = ((unsigned char *)&_func_rv_a)[i];
    unsigned char _func_rv_b_i = ((unsigned char *)&_func_rv_b)[i];
    klee_assert(_func_rv_a_i == _func_rv_b_i);
  }
  return 0;
}
```

图 4　用于等价性约束求解的产品程序示例

2.4　LLM 提示工程

在二进制代码提升与中间代码反编译阶段, LLM 承担着代码生成的关键角色. 为有效引导 LLM 正确执行代码理解与逆向生成任务, 提示词的设计至关重要. 本节将介绍本文所采用的提示词构建过程.

总体而言, BinDec 在使用 LLM 时采用迭代反馈机制. 该方法首先由 LLM 生成初始结果, 再借助外部工具对生成结果进行分析与反馈, 进而指导 LLM 对输出进行修正. 若经多轮修正后, LLM 的输出仍无法通过外部工具验证, 则放弃当前结果, 并利用 LLM 输出的随机性重新生成.

在具体每个阶段的 LLM 任务中, 本文采用了一种 LLM 辅助的提示词设计流程. 首先, 通过预实验选取若干典型代码样本, 并请求 LLM 生成面向特定任务的提示词. 例如, 提供若干汇编代码与 IR 代码的对应样本, 要求 LLM 为二进制代码提升任务生成相应提示词. 随后, 使用该提示词执行相应任务, 并基于人工检查与外部工具验证的结果, 总结生成内容中存在的问题, 反馈给 LLM 以指导其优化提示词. 通过多次交互迭代, 直至 LLM 所生成的提示词在预实验样本上的任务完成率趋于稳定.

针对本文中的代码生成与修复任务, 最终采用的提示词具有以下 3 项共同特征. 其一, 提出语法规范要求, 以引导 LLM 生成语法正确且可编译的代码. 其二, 强调语义保真约束, 要求特定函数在生成或修复后与原始输入的语义保持一致, 从而提高语义检查的通过率. 其三, 规范输出格式, 限制 LLM 仅输出代码及必要注释, 避免冗余解释, 在增强代码可读性的同时便于后续对生成代码的解析与提取. 前述 LLM 任务所采用的提示词如表 2 所示.

表 2　BinDec 各阶段任务使用的 LLM 提示词

LLM任务	提示词模板
二进制代码提升	作为资深软件逆向工程专家, 需将{asm_type}汇编代码转换为功能等价的 LLVM IR 代码, 确保: 1) 版本合规性: 严格遵循LLVM 13.0.1工具链规范, 禁用LLVM 14+新增语法 (如 opaque ptr), 禁用以llvm.*为前缀的优化伪指令 2) 语义保真度: `{func_name}`函数的功能与原始汇编代码完全一致, 包含必要的外部函数和全局变量的声明 3) 生成质量: 使用最合适的IR指令, 最小化冗余指令并保持合理的可读性 4) 输出格式: 仅输出提升后的LLVM IR代码, 允许注释但不要包含解释或其他额外文本

表 2 BinDec 各阶段任务使用的 LLM 提示词 (续)

LLM任务	提示词模板
中间代码错误修复	作为资深LLVM工具链专家, 需对从{asm_type}汇编提升得到的缺陷IR代码进行精准修复, 确保: 1) 版本合规性: 严格遵循LLVM 13.0.1工具链规范, 禁用LLVM 14+新增语法 (如 opaque ptr) 2) 语义保真度: 功能等价于原始汇编代码 (特别是`{func_name}`函数) 3) 修改最小化: 保持原始代码结构, 仅作必要修复 4) 输出格式: 仅输出修复后的LLVM IR代码, 允许注释但不要包含解释或其他额外文本
中间代码反编译	作为资深软件逆向工程与编程专家, 需将 LLVM IR 代码反编译为功能等价的 C 语言代码, 确保: 1) 语法合规性: 严格遵循C语言代码规范, 确保生成的代码可编译 2) 语义保真度: `{func_name}`函数的功能与原始LLVM IR代码完全一致, 包含必要的外部函数和全局变量的声明 3) 生成质量: 符合人类程序员的代码编写习惯, 最小化冗余代码并保持合理的可读性 4) 输出格式: 仅输出反编译后的C语言代码, 允许注释但不要包含解释或其他额外文本
源代码错误修复	作为资深软件逆向工程与编程专家, 需对从 LLVM IR 反编译得到的缺陷 C 代码进行精准修复, 确保: 1) 语法合规性: 严格遵循C语言代码规范, 确保生成的代码可编译 2) 语义保真度: 功能等价于原始LLVM IR代码 (特别是`{func_name}`函数) 3) 修改最小化: 保持原始代码结构, 仅作必要修复 4) 输出格式: 仅输出反编译后的C语言代码, 允许注释但不要包含解释或其他额外文本

2.5 函数级符号执行

为实现对单个函数的定向分析 (如测试用例生成或语义等价性验证), 需突破传统符号执行从程序入口 (main 函数) 开始的限制. 在第 2.2 和 2.3 节所述的任务中, 我们通过生成产品程序 (测试驱动程序), 对目标函数参数进行符号化. 函数级符号执行的关键在于, 如何在缺乏先验约束的情况下对参数进行有效的符号化建模, 以避免因过度近似而引发不必要的路径搜索. Ramos 等人[31]提出的欠约束符号执行框架 UC-KLEE, 采用惰性初始化的方式对参数进行动态符号化, 并允许用户通过手工定义不变式来约束符号空间. 反编译器测试框架 D-Helix[19]采用类似的惰性初始化策略对函数参数进行符号化建模. 然而, 为了实现对大量函数的自动化分析, D-Helix 并未为每个函数单独定义不变式约束, 这导致其在处理具有复杂数据结构的参数时, 执行效率受到显著影响. 为应对现实代码中复杂的参数类型 (特别是包含指针的嵌套结构体), 并在路径覆盖的完备性与执行效率之间取得平衡, 本文设计并实现了一种分层式预先符号化初始化策略. 该策略在符号执行引擎启动之前, 即一次性完成对参数结构的完整符号化构建. 具体而言, 本策略根据参数类型进行分层处理: 对于基本类型参数 (例如 int、char、float), 直接将其声明为符号化变量; 对于结构体参数, 则递归处理其每个字段——基本类型字段直接符号化, 对于指针字段的处理分为 3 步: 1) 为该指针字段本身分配一个具体的、有效的地址值, 确保其非空; 2) 为该地址分配一个由指针基类型决定大小的符号化内存对象; 3) 若指向结构体, 则对这块新内存递归应用本策略, 以符号化其内部字段. 为控制由此可能引发的路径状态空间爆炸, 递归过程受到深度限制 (如设定深度=3), 超限部分将被视为未初始化的符号字节序列. 此外, 对于顶层的指针类型参数, 其处理方式与结构体中的指针字段完全一致.

与现有的欠约束符号执行方法所采用的惰性初始化相比, 我们的方法通过预先初始化在符号执行开始前即构建了完整的符号输入结构, 实现上更为简单直接, 对于具有确定性、深度可控数据结构的函数, 能够确保路径探索的起点一致且可控.

3 实验设计

本节介绍实验设计情况, 包括研究问题、基准方法、实验数据、评估指标和实现细节.

3.1 研究问题

为了评估本文提出方法的有效性, 我们设计了以下 4 个问题进行研究.

问题 1: 与基线方法相比, BinDec 方法在 RISC-V 二进制代码的反编译任务上表现如何? 该问题的目的是评估 BinDec 方法的总体性能.

问题 2: 基于 LLM 迭代重试生成并修复代码的效果如何? 该问题的目的是讨论 BinDec 方法中所采用的迭代

式生成策略的时间和计算资源开销以及相应的算力成本.

问题 3: 基于符号执行的代码等价性检查准确性如何? 该问题的目的是探讨将符号执行技术应用于反编译任务时的可靠性.

问题 4: 与传统反编译器相比, BinDec 方法的失败模式有何异同? 该问题的目的是探究 BinDec 方法是否克服了传统反编译器的常见错误类型以及是否引入了新的错误类型.

3.2 基准方法

为评估本文所提出方法的有效性, 我们选取传统反编译工具作为主要基准进行比较. 在对 Ghidra、IDA Pro、angr 及 RetDec 等主流工具进行调研后, 发现仅有 Ghidra 与 IDA Pro 对 RISC-V 架构提供了较为完善的支持. 综合考虑工具的可获取性与集成开发复杂度, 我们最终选用 Ghidra 作为基准方法. Ghidra 首先将二进制代码提升为一种寄存器传输级中间表示 (P-code), 随后基于预设规则对 P-code 进行优化, 并借助启发式方法将其反编译为 C 代码. 需要说明的是, 尽管在二进制代码提升阶段我们使用了 Ghidra 的反汇编器组件, 但这并不影响将其反编译器组件作为基准方法的公平性. 此外, 目前尚无针对 RISC-V 架构反编译任务的专用深度学习方法, 为对本文方法中的程序分析部分进行对照评估, 我们构建了一个基于 LLM 的独立工作流作为另一基准方法 (称为 PureLLM), 其直接使用 LLM 根据汇编代码生成对应的反编译 C 代码.

3.3 实验数据

本文选取 ExeBench 作为基础数据集以构建实验所需的样本集合. ExeBench[32]是一个基于 GitHub 开源仓库构建的大规模代码数据集, 其中包含 450 万个可编译的 C 语言函数, 涵盖约 70 万个可执行函数. ExeBench 的特点在于其采用专门设计的自动化方法, 从真实代码环境中提取函数级片段, 并为部分样本函数生成了大量单元测试用例 (基于 3 组随机种子, 每组生成 10 个测试用例, 每个函数共包含 30 个测试用例). 这一特性为本文开展反编译代码的语义评估提供了重要支持.

为构建适用于本研究的实验数据集, 按照以下流程对基础数据进行处理. 首先, 对样本进行初步筛选. 由于本文提出的 BinDec 方法依赖于符号执行技术进行代码等价性检查, 该过程需使用函数的参数与返回值, 因此过滤掉无参数和无返回值的样本. 此外, 为避免极端函数长度对实验产生干扰, 我们根据函数字节长度剔除了长度最短与最长的各 5% 的样本. 这一处理基于以下考虑: 其一, 这些样本属于统计异常值 (如极短的无效桩函数或极长的冗余模板函数); 其二, 剔除它们可使评估更聚焦于典型函数, 从而提升结果的代表性和稳健性. 随后, 使用 Clang 将筛选后的源代码分别在 O0、O1、O2 和 O3 这 4 个优化级别下编译为 IR 代码, 形成中间代码样本集; 再进一步将 IR 代码编译为面向 RISC-V 平台的二进制程序. 之后, 借助 Ghidra 反编译工具对生成的二进制文件进行反编译, 以获取基准反编译代码. 需要说明的是, Ghidra 在处理部分样本时可能出现报错或长时间无响应的情况. 鉴于本研究主要聚焦于评估 BinDec 方法的反编译性能, 本文未对这类失败案例进行深入分析, 而是将其直接剔除, 仅保留可在限定时间内成功反编译的样本. 最后, 对数据集中所有源代码 (包括原始代码及由 Ghidra 反编译生成的代码) 进行统一格式化处理, 以消除代码风格差异对后续比较评估的潜在干扰. 经过上述处理, 最终共采用 16 543 个有效代码样本, 其详细统计信息如表 3 所示.

表 3 数据集样本统计特征 (字节)

样本类别	最小长度	平均长度	最大长度
源代码	614	771.74	1 062
中间代码	169	1 390.09	20 318
二进制代码	760	963.49	2 656

3.4 评估指标

本文提出的 BinDec 反编译方法, 旨在追求所生成的反编译代码与传统反编译器的输出具有相当准确性的同时, 能够提供更好的可读性. 为此, 我们参考 SLaDe[24]中所采用的评估方式, 从代码功能准确性和可读性两个方面

设计了评估指标.

在反编译代码的准确性评估中, 核心目标是验证反编译生成的代码与原始源代码在功能上是否等价. 由于程序输入空间通常是无限的, 一般采用抽象方法 (例如符号执行) 来验证功能等价性. 但在本文中, 符号执行技术已被整合进整体框架中, 因此不适合再作为外部评估手段. 考虑到我们所采用的数据集本身提供了高质量的测试用例集合 (每个样本包含 30 个测试用例), 本文通过评估反编译代码在集成单元测试上的表现来衡量其质量, 并据此定义了一系列量化指标, 具体评估流程如下所述. 首先, 将反编译得到的函数级代码单独编译为对象模块, 该阶段是否成功编译称为函数可编译性 (RCf). 若编译过程中无错误产生, 则认为该函数满足 RCf 要求. 接下来, 将单元测试模板程序与此函数模块进行链接, 由于链接阶段仍可能出现错误 (例如函数签名不一致或调用约定不匹配), 此时的可编译性称为程序可编译性 (RCp). 若链接成功, 则生成可执行的测试程序. 随后, 运行所生成的测试程序, 若程序能够正常执行且未发生运行时错误 (例如内存非法访问或异常崩溃), 则认为该程序具备可执行性 (RE). 最后, 在程序可执行的基础上, 统计其在所有测试输入上的输出正确率, 作为评估功能一致性的关键指标, 即测试通过率 (IOAcc). 该通过率基于数据集中每个样本对应的 30 个测试用例计算, 具体定义为通过用例数占总用例数的比例. 综上所述, 函数可编译性关注函数模块本身的编译可行性, 程序可编译性侧重于模块与测试框架的链接能力, 可执行性反映程序运行时的稳定性, 而测试通过率则从输入输出行为角度评估功能正确性. 这 4 个指标共同构成对反编译结果准确性的多层次评估体系.

对于反编译代码的可读性, 其便于人类理解的目标是一个主观概念, 直接进行大规模人工评估成本高昂且难以复现. 因此, 在研究中广泛采用文本相似度作为可读性的可量化代理指标, 其基本假设是: 由经验丰富的开发者编写、来自真实项目的源代码 (即本文数据集中的参考源代码) 具有较高的可读性. 因此, 反编译代码与参考源代码在文本上的相似性可以在一定程度上反映其是否遵循了常见的编码惯例 (如变量命名、代码结构), 从而间接衡量其可读性. 具体来说, 我们采用 BLEU[33]与编辑相似度 (edit similarity, ES)[24]作为评价指标. BLEU 通过比较模型生成的文本与参考文本之间的 n-gram 重叠程度来量化序列层面的共性. ES 则通过 Levenshtein 编辑距离量化两个序列间的差异, 操作次数越少则相似度越高. 文本相似度可用于衡量生成的反编译代码与原始源代码之间的表面相似性, 从而间接反映其是否具备与原始代码相近的可读性水平.

关于评估指标的局限性我们将在实验效度威胁部分 (第 5.2 节) 进行讨论.

3.5 实现细节

本文实验主要使用 Python 3.10 进行开发, 所有实验在一台运行 Ubuntu 22.04 操作系统的服务器上开展, 其硬件配置为 64 核 CPU (Intel(R) Xeon(R) Gold 5218 CPU @ 2.30 GHz), 192 GB 内存.

为提取函数级汇编指令序列及相关函数信息, 我们基于 Ghidra (版本 11.4) 提供的 Python API 开发了定制插件, 以实现反汇编功能. 为提取函数级 IR 代码片段, 我们基于 LLVM (版本 13.0.1) 的 API 实现了一个模块插件 (llvm::ModulePass). 在验证代码语义一致性时, 使用 KLEE[34] (版本 3.1) 作为符号执行引擎, 并搭配 Z3 (版本 4.9.1) 作为 SMT 求解器. 为在 x86-64 架构的测试机上运行 RISC-V 架构的可执行程序, 采用 QEMU-RISCV64 (版本 6.2.0) 作为二进制模拟器. 为实现对 RISC-V 二进制程序中特定函数的注入, 使用 LUI 与 JALR 指令组合对待替换函数所在内存位置进行指令覆盖, 从而将原始函数调用重定向至替换位置. 该指令组合支持 32 位地址空间内的绝对地址跳转, 我们在使用 QEMU-RISCV64 时将虚拟地址空间大小限制为 1 GiB, 以兼顾该指令组合的跳转空间限制和人工分析的便利性. 在 LLM 方面, 直接调用公有云提供的大模型 API 服务, 使用的模型为 DeepSeek-V3.

4 实验结果与分析

为了评估本文所提出的 BinDec 方法的效果, 本节详细报告设计的实验问题, 并对实验结果进行分析.

4.1 问题 1: 与基线方法相比, BinDec 方法在 RISC-V 二进制代码的反编译任务上表现如何?

针对问题 1, 我们依据第 3 节所述的实验设置进行了验证. 在二进制代码提升与中间代码反编译两个阶段中,

最大重试次数分别设定为 10 次和 5 次. 实验结果如表 4 所示, 我们计算了第 3.4 节所建立的各项评估指标在实验样本上的平均值.

表 4　BinDec 与基准方法的反编译结果对比 (%)

方法	RCf	RCp	RE	IOAcc	BLEU	ES
Ghidra	67.10	67.10	67.10	62.98	7.25	17.42
PureLLM	60.92	56.70	51.32	37.50	15.27	24.23
BinDec	71.93	71.78	71.74	63.18	20.44	31.41
BinDec (O0)	70.41	70.09	69.94	60.91	24.56	33.42
BinDec (O1)	71.59	71.43	71.43	62.93	19.16	30.51
BinDec (O2)	72.69	72.54	72.54	63.28	19.28	30.93
BinDec (O3)	72.79	72.67	72.67	65.47	19.00	30.90

实验结果表明, 本文提出的 BinDec 方法在反编译正确性方面达到了与传统反编译器 Ghidra 相当的水平, 同时在代码可读性方面显著优于 Ghidra, 其可读性指标为 Ghidra 的两倍以上. 具体而言, BinDec 在 BLEU 和 ES 两个指标上的表现分别是 Ghidra 的 2.8 倍和 1.8 倍. 这一结果说明, BinDec 所生成的反编译代码与数据集中原始真实代码的相似度更高, 体现出 LLM 在代码生成任务中的优势. 由于 LLM 在海量代码数据上进行过预训练, 它能够生成更符合人类阅读习惯的高级语言代码, 从而规避传统反编译方法因缺乏上下文而难以恢复高可读性代码的缺陷. 此外, 在功能性指标方面, BinDec 在函数可编译率 (RCf)、程序可编译率 (RCp)、可执行率 (RE) 与测试通过率 (IOAcc) 上分别比 Ghidra 高出 7.2%、6.9%、6.9% 和 0.3%. 相比之下, PureLLM 作为一种仅基于 LLM 的基线方法, 在所有代码相关指标上均显著落后于 BinDec 和 Ghidra. 这一对比凸显了 BinDec 方法中引入的符号执行技术对提升反编译代码可靠性的重要作用. 进一步分析可以发现, BinDec 在 RCf、RCp 与 RE 这 3 个递进指标上呈逐级下降趋势, 而 Ghidra 在这些指标上保持一致. 这 3 个指标分别反映了单个函数的语法正确性、多个函数组合成完整程序的语法正确性以及最终生成程序的实际运行能力. 尽管 BinDec 在函数可编译率上领先 Ghidra 7.2%, 但其测试通过率仅高出 0.3%, 说明 BinDec 目前采用的可靠性保障机制仍存在一定局限性. 我们认为, BinDec 方法的表现主要受以下两方面因素的影响: 1) 外部函数简化模型在重建完整程序时引入了不确定性; 2) 基于符号执行的代码语义一致性检查方法可能存在漏报情形, 从而影响最终生成代码的可靠性.

与此同时, 我们展示了采用 BinDec 方法生成的反编译代码在不同编译优化级别 (O0–O3) 下的分组统计结果. 分析表明, BinDec 方法在 O0 优化级别下表现出最佳的可读性, 而在 O3 级别下则获得了最优的代码功能完整性指标. 这一结果反映出在反编译代码的可读性与功能正确性之间存在一定的权衡关系, 即较高的功能完整性往往伴随着可读性的下降, 反之亦然. 我们认为, 该现象与不同优化级别下编译器生成的二进制代码特征密切相关. 在较低优化级别 (如 O0) 下, 编译器不会进行深度优化, 代码中保留了较多的中间变量和原始控制流结构, LLM 能够利用这些信息提升生成代码的可读性. 然而, 由于未进行代码体积优化, 此类二进制代码通常较为冗长, 增加了 LLM 处理过程中的上下文负担, 导致生成的反编译代码可能出现语义准确性不足的问题. 相反, 在较高优化级别 (如 O3) 下, 编译器会采取一系列激进优化策略, 例如函数内联、循环优化和冗余代码消除等, 使得生成的二进制代码更为紧凑与高效. 此类优化虽然削弱了代码中的可读性结构信息, 但同时降低了 LLM 理解代码时所需的上下文复杂度, 从而有助于生成功能更完备的反编译结果.

进一步地, 为了定量探究函数复杂度对 BinDec 方法性能的影响, 我们参考了 DecLLM[28], 将二进制代码的长度作为函数复杂度的代理指标, 并计算其与测试通过率的斯皮尔曼相关系数, 结果如表 5 所示. 结果表明, BinDec 方法的性能随着函数复杂度的增加呈现显著的弱负相关, 这与前文的定性分析结论一致, 表明 LLM 在处理长上下文时幻觉问题会加剧. 这一发现凸显了本文所采用的形式化验证策略的重要性, 该策略作为一种强约束, 能有效过滤模型的不可靠输出, 从而保障生成结果的可靠性. 相比之下, Ghidra 的性能则与函数复杂度无显著相关, 符合其基于规则的方法特性. 这揭示了数据驱动方法与基于规则的方法在不同场景下的互补性. 综上, 对复杂度的分析指

出了当前方法的局限性, 也为未来工作指明了方向, 如引入更精细的复杂度度量或针对高复杂度函数的专项优化, 以进一步提升方法的鲁棒性.

表 5 函数复杂度与测试通过率的相关性

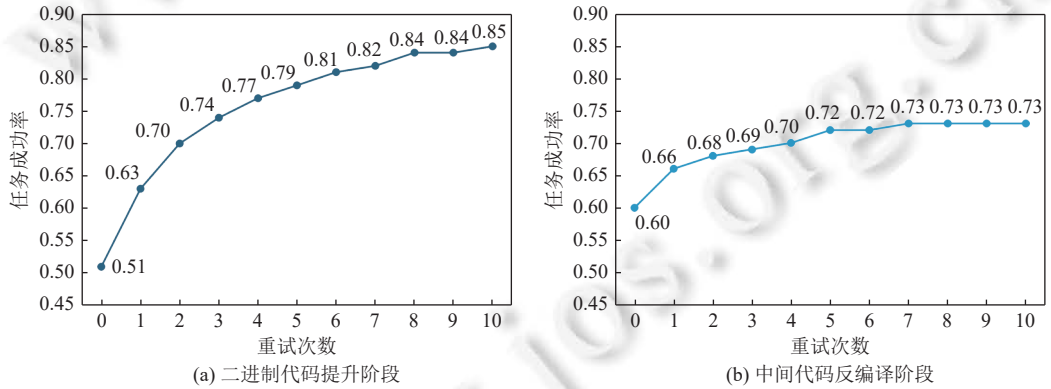
方法	相关系数 (ρ)	显著性 (p)
Ghidra	−0.125	>0.05
BinDec	−0.393	<0.05

针对问题 1 的总结: 本文提出的 BinDec 方法能够生成在功能正确性与代码可读性两方面均优于传统反编译器 Ghidra 的反编译结果. 尤其是在对经过高优化级别编译的二进制代码进行反编译时, 该方法所展现的综合优势更为显著.

4.2 问题 2: 基于 LLM 迭代重试生成并修复代码的效果如何?

本文提出的 BinDec 方法采用迭代重试策略对 LLM 生成的初始代码进行错误修复, 以提升反编译结果的可靠性. 由于每一次迭代过程均包含额外的 LLM 查询、代码编译与符号执行操作, 这些步骤会引入显著的时间开销与计算资源消耗. 因此, 迭代过程的收敛效率成为影响方法实用性的关键因素. 针对问题 2, 本研究重点对迭代重试过程的收敛性进行分析. 在实验设置中, 我们将二进制代码提升与中间代码反编译两个阶段的最大重试次数统一设定为 10 次, 并完整记录每一轮重试所产生的代码版本、LLM 查询的输入与输出数据. 在此基础上, 对所有重试步骤中生成的数据进行统计评估.

首先, 我们对重试任务的成功率进行了统计分析. 该成功率定义为在特定重试次数下生成语义一致代码的样本比例, 统计结果如图 5 所示. 实验结果表明, 在二进制代码提升与中间代码反编译两个阶段中, 随着重试次数的增加, 任务成功率均呈现上升趋势, 并在达到一定重试次数后逐渐趋于稳定. 具体而言, 在二进制代码提升阶段, 首次重试的成功率为 51%, 即 51% 的样本无需重试即可生成语义一致的中间代码; 当重试次数达到 8 次时, 成功率趋于稳定, 维持在 84%–85% 之间. 在中间代码反编译阶段, 首次成功率为 60%, 即 60% 的样本能够一次性生成语义一致的 C 代码; 当重试次数达到 5 次时, 成功率趋于稳定, 保持在 72%–73% 的范围内. 两个阶段所呈现的不同趋势说明, 二进制代码提升过程更依赖于重试机制, 并且能够从中获得更为显著的性能提升.



(a) 二进制代码提升阶段 (b) 中间代码反编译阶段

图 5 不同重试次数下的任务成功率

其次, 我们针对重试任务中编译器和符号执行引擎的累计调用次数进行了统计分析, 结果如图 6 所示. 在每次重试过程中, 生成的代码必须经过编译器调用以完成编译, 而符号执行引擎仅在编译成功之后才会被调用. 实验结果显示, 在二进制代码提升任务中, 随着重试次数的增加, 编译器调用次数与符号执行调用次数之间存在显著差异, 这一现象表明 LLM 生成的中间代码在可编译性方面仍面临较大挑战, 重试过程主要用于修正编译错误. 相比之下, 在中间代码反编译任务中, 编译器调用次数与符号执行调用次数基本保持一致, 并呈现近似线性的增长趋势, 说明 LLM 生成的 C 源代码具有较好的可编译性, 但其语义一致性仍是需要重点解决的问题. 此外, 我们对两

阶段任务中的平均编译和符号执行耗时进行了统计 (如表 6 所示), 可见符号执行所需的时间显著高于编译, 我们将此差距主要归因于符号执行在路径爆炸和约束求解方面的固有限制.

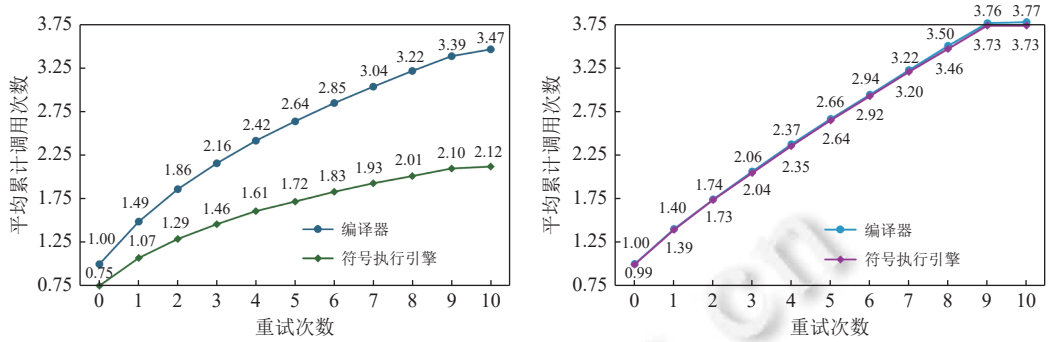


图 6 不同重试次数下的编译器和符号执行引擎平均累计调用次数

表 6 对单个函数进行反编译的平均成本

任务阶段	编译时间 (s)	符号执行时间 (s)	LLM输入 (元)	LLM输出 (元)
二进制代码提升	0.018	2.797	0.018	0.030
中间代码反编译	0.443	8.040	0.010	0.012

最后, 我们统计了重试任务中 LLM 查询的平均累计调用数据量, 结果如图 7 所示. 可以看出, 在两个任务中, LLM 查询的调用数据量均呈近似线性增长趋势, 且二进制代码提升任务所使用的输入输出调用数据量明显高于中间代码反编译任务, 表明前者需要更多的计算资源支持. 本实验中 LLM 调用基于公有云 API 实现, 其服务费用根据输入和输出的 token 数量进行计算, 因此上述差异直接反映了实际应用中的经济成本差异 (针对单个函数的平均成本如表 6 所示).

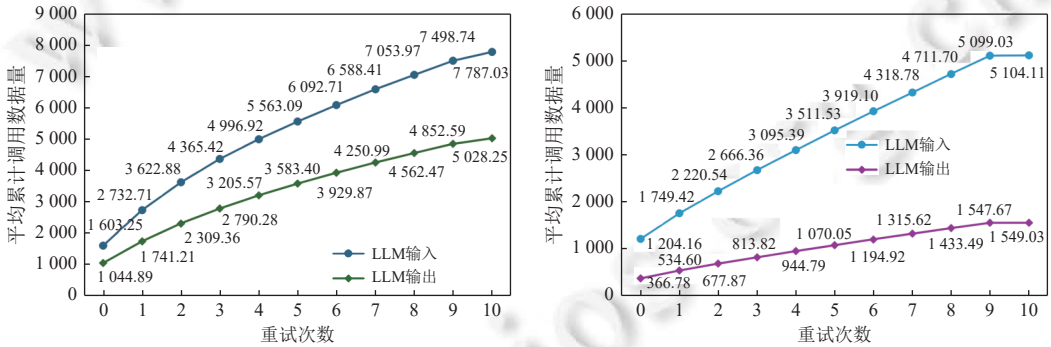


图 7 不同重试次数下的 LLM 查询平均累计调用数据量

针对问题 2 的总结: 本文所提出的 BinDec 方法依赖的迭代重试机制整体上是有效的, 且在二进制代码提升阶段所带来的效果提升更为显著. 然而, 这一阶段也消耗了更多的 LLM 计算资源. 在实际应用中, 需综合考虑任务成功率与资源使用量之间的平衡, 以实现经济效益与性能的优化.

4.3 问题 3: 基于符号执行的代码等价性检查准确性如何?

本文提出的 BinDec 反编译方法, 其核心在于引入符号执行技术对 LLM 输出的代码进行语义一致性检查, 以提升生成代码的可靠性. 因此, 验证基于符号执行的代码等价性检查的准确性具有关键意义. 我们在第 3 节所述方

法构建的数据集基础上, 选取了 2000 个样本, 构造了一个专门用于验证等价性检查准确性的数据集. 该数据集的构建过程如下: 首先随机选取 1000 个由不同源代码编译生成的样本作为参考代码; 随后, 对每一个参考代码, 随机选取来自同一源代码但在不同编译优化级别下生成的样本, 作为语义一致的正样本; 最后, 对每一个参考代码, 再随机选取来自不同源代码且在任意编译优化级别下生成的样本, 作为语义不一致的负样本. 我们在二进制代码与中间代码两个层面上, 依据本文所采用的等价性检查流程, 分别在该数据集上进行了测试, 并使用二分类评估指标 (精确率、召回率与 $F1$ 分数) 对实验结果进行评估, 具体结果如表 7 所示. 实验表明, 基于符号执行的代码等价性检查方法在两类代码上均取得了较高的召回率 (二进制代码为 99.71%, 中间代码为 98.56%), 说明该方法能够有效识别语义不一致的代码对. 然而, 其精确率相对较低 (二进制代码为 86.10%, 中间代码为 91.38%), 表明存在一定比例的将语义一致代码误判为不一致的情况. 进一步对比二进制代码与中间代码上的检查效果, 可发现基于符号执行的方法在二进制代码上表现更为严格, 但也产生了更多误报. 我们认为这一现象与两个阶段代码的检查方式差异有关. 对于中间代码, BinDec 方法构建了统一的符号模型, 并通过约束求解完成一致性判断; 而在二进制代码层面, 该方法则基于符号执行生成测试用例, 并在实际程序环境中执行这些用例. 由于真实执行环境对运行条件的要求更为严苛, 容易出现运行异常或失败, 从而导致检查过程更为严格, 误报率相应升高.

表 7　代码等价性检查准确性结果 (%)

检查对象	精确率	召回率	$F1$
二进制代码	86.10	99.71	92.41
中间代码	91.38	98.56	94.83

针对问题 3 的总结: 本文所提出的基于符号执行的方法可以实现低漏报的代码等价性检查, 因此其可以作为保证 LLM 生成代码质量的可靠工具.

4.4　问题 4: 与传统反编译器相比, BinDec 方法的失败模式有何异同?

为深入探究本文所提出的 BinDec 方法与传统反编译器 Ghidra 在失败模式上的异同, 我们对两者在实验测试集上产生的失败案例进行了系统的归纳与对比分析. 首先, 我们依据错误发生的阶段和外在表现, 构建了一个双层分类框架: 第 1 层级将失败划分为编译错误 (即反编译代码无法通过编译) 与执行错误 (即程序可编译但运行结果异常). 在第 2 层级中, 我们进一步根据编译器报错信息 (如语法错误、类型不匹配、未定义符号等) 对编译错误进行细分; 对于执行错误, 则根据其异常行为的本质, 划分为与程序稳定性有关的内存错误 (如段错误、内存泄漏) 和与语义等价有关的逻辑错误 (如输出不符、无限循环). 基于此框架, 我们对所有失败案例进行了人工复核与归类, 其统计结果如表 8 所示. 该分类结果表明两种方法在错误类型分布上的总体差异, 即 BinDec 方法克服了传统反编译器 Ghidra 中的典型错误, 并且未引入新的错误类型. 而 BinDec 方法中仍然存在的错误 (例如未定义变量) 相较其他错误更容易手工解决, 这表明 BinDec 方法生成的反编译代码具有更高的重用价值.

表 8　反编译代码的典型错误归纳 (%)

错误分类	子类别	BinDec	Ghidra
编译错误	未定义变量	90	60
	未定义类型	0	13
	未定义函数	0	6
	不兼容的类型	0	3
	语法错误	5	10
	其他	5	8
执行错误	内存错误	100	90
	逻辑错误	0	10

针对问题 4 的总结: 本文提出的 BinDec 方法在失败模式上实现了对传统反编译器中复杂错误的规避, 其失败案例呈现出更高的可修正性, 这为其在软件逆向工程中的实用价值提供了实证支持.

5 讨 论

本节讨论本文所提 BinDec 方法的局限性以及所开展实验方法的效度威胁.

5.1 局限性

本文提出的 BinDec 方法存在 3 个主要局限性, 分别涉及符号执行过程中的模型简化策略、对 LLM 错误结果的处理方式以及等价性检查策略.

首先, 在针对单个函数进行符号执行时, 为兼顾准确性与效率, 我们采用了 3 种模型简化策略以支持代码等价性检查. 其一, 我们仅对函数输入参数至返回值之间的传递关系进行建模, 而忽略函数执行过程中可能产生的副作用. 然而, 真实应用场景中存在大量不符合该假定的函数, 例如无参数函数、无返回值函数以及功能主要依赖于副作用的函数. 这一限制使本文方法仅能适用于同时具有输入参数和明确返回值的函数. 其二, 为实现独立针对单个函数的符号执行, 我们对其中的外部调用进行了简化处理, 具体表现为将所有外部调用的功能定义为“返回所有输入参数的最低字节之和”. 该策略虽有效避免了符号值传递过程的中断, 但在某些情况下可能导致多个不同外部调用被误建模为相同逻辑, 从而引发符号执行中的路径合并错误. 其三, 我们在对函数参数进行符号化建模时, 限制了对复杂类型参数的递归展开深度, 这种策略在处理深度不确定或递归定义的动态数据结构 (例如, 长度可变的链表) 时存在局限, 因为预设的深度限制可能无法覆盖所有可能的情况.

其次, 当前 BinDec 方法无法充分复用 LLM 生成的错误结果. 在 BinDec 的流程中, 若 LLM 生成的代码存在编译错误, 框架会尝试要求 LLM 对其进行修复; 然而, 若生成的代码虽可编译但与预期语义不一致, 则当前做法会直接丢弃该结果. 一种可行的改进思路是引导 LLM 基于语义不一致的代码进行局部修复, 而非完全重新生成. 但由于难以根据符号执行的结果精确定位导致语义偏差的具体代码位置, 因此无法构建有效的提示词以指导 LLM 完成语义层面的修复.

最后, 在二进制代码提升阶段的等价性检查中, 我们当前依赖于基于符号执行生成的测试用例进行动态测试, 而未采用完全符号化的约束求解方法. 这主要是由于现阶段面向二进制代码的符号执行技术仍存在一定局限性. 现有成熟工具如 BINSEC[35], 需先将二进制代码提升至其自定义中间表示 DBA 再进行符号执行, 本质上仍依赖于中间表示转换. 而 BinSym[36]虽支持直接在二进制层面进行符号执行, 但目前仅适用于 RISC-V 架构, 无法满足本文构建架构无关反编译框架的目标. 因此, 我们采用符号执行生成高覆盖率测试用例作为现阶段可行性较高的验证方案, 其可靠性高度依赖于测试用例对代码行为的覆盖程度. 尽管如此, 本方法仍具有一致性优势, 即两个阶段的验证均基于同一符号执行引擎 KLEE 实现, 在工具层面保持了统一, 也避免了引入额外中间表示或架构相关工具所带来的复杂性. 未来, 随着二进制符号执行技术的进一步发展, 采用统一的、完全符号化的端到端验证框架, 将有望进一步提升本方法的整体可靠性与形式化保证强度.

综上所述, 尽管实验结果表明将 LLM 与符号执行相结合能够有效提升二进制代码反编译能力, 当前方法在 LLM 与符号执行引擎之间的交互机制仍较为薄弱, 导致其在场景适应性与能效比方面存在明显局限. 未来的研究工作需探索更有效的协同策略, 以加强 LLM 与符号执行技术之间的互补与融合, 从而充分发挥二者各自的优势.

5.2 实验效度威胁

本节讨论本文实验方法可能面临的效度威胁. 效度威胁是指实验过程中可能对结论有效性产生不利影响的因素, 主要包括构建效度和外部效度两方面.

构建效度关注于度量指标是否能够有效反映所研究的核心概念. 在本研究中, 具体指所采用的准确性和可读性评估指标是否能够准确衡量反编译代码的质量. 对于准确性, 当前的评估流程要求代码必须先成功编译才能进行后续的可执行性验证和单元测试. 这意味着, 一个语义上几乎完全正确但无法通过编译的反编译结果 (例如, 细微的语法错误或不符合特定编译器规范) 将在所有功能维度上得分为 0. 这种“一票否决”机制可能导致评估体系过于严格, 尤其可能会对 Ghidra 这类代码格式并非完美无缺的工具产生不公平的低估. 尽管存在此威胁, 但我们认为将可编译性作为基础要求符合工程实践. 不过需要说明的是, 这一设计意味着我们的功能评估主要反映代码

的工程可用性 (例如可被项目直接集成), 而非纯粹的语义正确性. 此外, 单元测试正确率依赖于数据集中测试用例的功能覆盖完备性, 尤其是 ExeBench 所构建的测试集只关注输入输出的匹配情况. 为验证单元测试集的可靠性, 我们在审查评估 ExeBench 数据集中单元测试构造方法论的基础上, 在最终实验数据中随机抽取 100 个样本进行人工复核, 结果未发现大范围的、隐蔽的逻辑不一致问题, 这表明单元测试正确率作为评估指标是有效的. 对于可读性, 本文选择文本相似度 (BLEU 和 ES) 作为其代理指标, 由于文本相似度主要衡量代码的表面特征, 因此对于功能语义完全正确但实现风格 (如变量命名、代码格式) 与参考代码不同的反编译结果, 其评分可能不公平地偏低. 鉴于此, 我们在预实验中对可读性进行了小规模的人工评估, 斯皮尔曼相关性分析表明, 人工评分与文本相似度 (采用 BLEU 和 ES 的均值) 得分呈显著中等程度正相关 (ρ=0.53, p<0.05), 即将文本相似度作为可读性的代理指标在本文采用的数据集上是有效的. 总之, 尽管本文采用的评估指标存在其固有的局限性, 但通过前文中详述的验证措施, 这些指标在实验中能够有效地服务于其主要目的, 即为各种反编译方法的输出提供一个公平、一致且可复现的排序依据. 因此, 该评估体系能够稳健地支撑本文的比较分析.

外部效度关注于实验结论的泛化能力, 具体包括本文所提方法是否能够推广至其他指令集架构的二进制反编译任务, 以及是否可适配于其他 LLM 进行代码生成与修复. 首先, BinDec 方法依赖于现有反汇编工具来将 RISC-V 二进制程序转换为汇编指令序列, 再输入后续流程. 与反编译器不同, 反汇编器在不同架构上的技术成熟度较为一致, 并且根据本文实验测试, LLM 对不同架构汇编代码的理解能力也较为接近. 因此, 我们认为 BinDec 方法可扩展至其他指令集架构的反编译任务. 其次, 尽管本文选用当前主流模型 DeepSeek-V3 作为实验所用的 LLM, 但现有许多 LLM 均面向代码任务进行了专门训练, 并具备类似的交互接口与代码生成能力. 因此我们推断, 替换为其他 LLM 仍可顺利完成 BinDec 工作流中的各项任务. 此外, 由于本文采用了符号执行技术对 LLM 输出结果进行验证与约束, 因此在不同模型下该方法仍能保持相对较高的可靠性.

6 总 结

反编译是信息安全和软件工程领域中的一项重要技术. 传统反编译器通常难以有效恢复在编译过程中丢失的高级语义信息, 因此大多仅能作为资深工程人员的辅助工具. 此外, 由于传统方法依赖大量人工定义的专家规则, 其完善过程需要投入显著的开发工作量, 导致对新兴硬件架构 (如 RISC-V) 的支持往往存在滞后. 随着深度学习技术的发展, 研究者开始尝试将反编译任务建模为代码翻译问题, 并构建相应的反编译模型. 在 LLM 取得突破后, 部分研究开始利用其强大的代码理解与生成能力来辅助反编译. 然而, 由于 LLM 本身存在不确定性与幻觉现象, 其输出结果的可靠性难以保证. 针对上述问题, 本文提出了一种面向 RISC-V 架构二进制代码的反编译框架 BinDec, 该框架将 LLM 技术与程序分析技术深度融合. BinDec 方法在充分利用 LLM 代码生成能力的基础上, 引入符号执行技术以验证和增强生成结果的可靠性, 从而实现更可信的反编译. 与现有反编译器相比, BinDec 方法所产生的代码具有更高的可读性与规范性, 有助于降低工程人员的代码分析负担, 甚至可直接将反编译结果用于重编译及新的软件工程项目.

尽管当前 LLM 在代码相关任务中表现出色, 但其不可解释性与输出不稳定性限制了其在复杂任务中的进一步应用. 本文研究了如何利用既有程序分析技术提升 LLM 输出的可靠性, 并验证了该策略在支持 RISC-V 等新兴指令集架构的系统软件开发中的可行性. 未来的研究工作将致力于探索更高效的 LLM 与程序分析协同机制, 以充分发挥双方优势, 并支持更复杂的反编译及其他软件工程任务.

References

[1] Dinesh S, Burow N, Xu DY, Payer M. RetroWrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 1497–1511. [doi: 10.1109/SP40000.2020.00009]

[2] Nagy S, Nguyen-Tuong A, Hiser JD, Davidson JW, Hicks M. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In: Proc. of the 30th USENIX Security Symp. USENIX Association, 2021. 1683–1700.

[3] Lee J, Avgerinos T, Brumley D. TIE: Principled reverse engineering of types in binary programs. In: Proc. of the 18th Network and Distributed System Security Symp. San Diego: The Internet Society, 2011.

[4] Lacomis J, Yin PC, Schwartz E, Allamanis M, Le Goues C, Neubig G, Vasilescu B. DIRE: A neural approach to decompiled identifier naming. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 628–639. [doi: 10.1109/ASE.2019.00064]

[5] Zhang Z, Ye YP, You W, Tao GW, Lee WC, Kwon Y, Aafer Y, Zhang XY. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In: Proc. of the 2021 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2021. 813–832. [doi: 10.1109/SP40001.2021.00051]

[6] Pei KX, Guan J, Broughton M, Chen ZT, Yao SC, Williams-King D, Ummadisetty V, Yang JF, Ray B, Jana S. StateFormer: Fine-grained type recovery from binaries using generative state modeling. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 690–702. [doi: 10.1145/3468264.3468607]

[7] Chen QB, Lacomis J, Schwartz EJ, Goues CL, Neubig G, Vasilescu B. Augmenting decompiler output with learned variable names and types. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX Association, 2022. 4327–4343.

[8] Katz DS, Ruchti J, Schulte E. Using recurrent neural networks for decompilation. In: Proc. of the 25th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Campobasso: IEEE, 2018. 346–356. [doi: 10.1109/SANER.2018.8330222]

[9] Fu C, Chen HL, Liu HL, Chen XY, Tian YD, Koushanfar F, Zhao JS. Coda: An end-to-end neural program decompiler. In: Proc. of the 33rd Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2019. 333. [doi: 10.5555/3454287.3454620]

[10] de Moura L, Bjørner N. Satisfiability modulo theories: Introduction and applications. Communications of the ACM, 2011, 54(9): 69–77. [doi: 10.1145/1995376.1995394]

[11] Chang TY, Chen SZ, Fan GD, Feng ZY. A self-iteration code generation method based on large language models. In: Proc. of the 29th Int'l Conf. on Parallel and Distributed Systems (ICPADS). Ocean Flower Island: IEEE, 2023. 275–281. [doi: 10.1109/ICPADS60453.2023.00049]

[12] Dong YH, Jiang X, Jin Z, Li G. Self-collaboration code generation via ChatGPT. ACM Trans. on Software Engineering and Methodology, 2024, 33(7): 189. [doi: 10.1145/3672459]

[13] Huang T, Sun ZH, Jin Z, Li G, Lyu C. Knowledge-aware code generation with large language models. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Program Comprehension (ICPC). Lisbon: ACM, 2024. 52–63. [doi: 10.1145/3643916.3644418]

[14] Jiang X, Dong YH, Wang LC, Fang Z, Shang QW, Li G, Jin Z, Jiao WP. Self-planning code generation with large language models. ACM Trans. on Software Engineering and Methodology, 2024, 33(7): 182. [doi: 10.1145/3672456]

[15] Fan ZY, Gao X, Mirchev M, Roychoudhury A, Tan SH. Automated repair of programs from large language models. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1469–1481. [doi: 10.1109/ICSE48619.2023.00128]

[16] Jin M, Shahriar S, Tufano M, Shi X, Lu S, Sundaresan N, Svyatkovskiy A. InferFix: End-to-end program repair with LLMs. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1646–1656. [doi: 10.1145/3611643.3613892]

[17] Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 919–931. [doi: 10.1109/ICSE48619.2023.00085]

[18] Yuan ZQ, Liu MW, Ding SJ, Wang KX, Chen YX, Peng X, Lou YL. Evaluating and improving ChatGPT for unit test generation. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 76. [doi: 10.1145/3660783]

[19] Zou MQ, Khan A, Wu RY, Gao H, Bianchi A, Tian D. D-Helix: A generic decompiler testing framework using symbolic differentiation. In: Proc. of the 33rd USENIX Security Symp. Philadelphia: USENIX Association, 2024. 397–414.

[20] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010. [doi: 10.5555/3295222.3295349]

[21] Hosseini I, Dolan-Gavitt B. Beyond the C: Retargetable decompilation using neural machine translation. In: Proc. of the 2022 Workshop on Binary Analysis Research (BAR). San Diego: The Internet Society, 2022. 1–11. [doi: 10.14722/bar.2022.23009]

[22] Al-Kaswan A, Ahmed T, Izadi M, Sawant AA, Devanbu P, van Deursen A. Extending source code pre-trained language models to summarise decompiled binaries. In: Proc. of the 30th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Taipa: IEEE, 2023. 260–271. [doi: 10.1109/SANER56733.2023.00033]

[23] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[24] Armengol-Estapé J, Woodruff J, Cummins C, O'Boyle MFP. SLaDe: A portable small language model decompiler for optimized assembly. In: Proc. of the 2024 IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO). Edinburgh: IEEE, 2024. 67–80. [doi: 10.1109/CGO57630.2024.10444788]

[25] Melo LTC, Ribeiro RG, Guimarães BCF, Pereira FMQ. Type inference for C: Applications to the static analysis of incomplete programs. ACM Trans. on Programming Languages and Systems, 2020, 42(3): 15. [doi: 10.1145/3421472]

[26] Jiang N, Wang CX, Liu K, Xu XZ, Tan L, Zhang XY, Babkin P. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. In: Proc. of the 13th Int'l Conf. on Learning Representations. Singapore: OpenReview.net, 2025.

[27] Xu XZ, Zhang Z, Su ZA, Huang ZY, Feng SW, Ye YP, Jiang N, Xie DN, Cheng SY, Tan L, Zhang XY. Unleashing the power of generative model in recovering variable names from stripped binary. In: Proc. of the 32nd Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2025. 1–18. [doi: 10.14722/ndss.2025.240276]

[28] Wong WK, Wu DY, Wang HJ, Li ZJ, Liu ZB, Wang S, Tang QY, Nie S, Wu S. DecLLM: LLM-augmented recompilable decompilation for enabling programmatic use of decompiled code. Proc. of the ACM on Software Engineering, 2025, 2(ISSTA): ISSTA081. [doi: 10.1145/3728958]

[29] Hu PW, Liang RG, Chen K. DeGPT: Optimizing decompiler output with LLM. In: Proc. of the 31st Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2024. 1–16. [doi: 10.14722/ndss.2024.24401]

[30] Glock J, Pichler J, Pinzger M. PASDA: A partition-based semantic differencing approach with best effort classification of undecided cases. Journal of Systems and Software, 2024, 213: 112037. [doi: 10.1016/j.jss.2024.112037]

[31] Ramos DA, Engler D. Under-constrained symbolic execution: Correctness checking for real code. In: Proc. of the 24th USENIX Security Symp. Washington: USENIX Association, 2015. 49–64.

[32] Armengol-Estapé J, Woodruff J, Brauckmann A, de Souza Magalhães JW, O'Boyle MFP. ExeBench: An ML-scale dataset of executable C functions. In: Proc. of the 6th ACM SIGPLAN Int'l Symp. on Machine Programming. San Diego: ACM, 2022. 50–59. [doi: 10.1145/3520312.3534867]

[33] Papineni K, Roukos S, Ward T, Zhu WJ. BLEU: A method for automatic evaluation of machine translation. In: Proc. of the 40th Annual Meeting of the Association for Computational Linguistics. Philadelphia: ACL, 2002. 311–318. [doi: 10.3115/1073083.1073135]

[34] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224. [doi: 10.5555/1855741.1855756]

[35] David R, Bardin S, Ta TD, Mounier L, Feist J, Potet ML, Marion JY. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In: Proc. of the 23rd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). Osaka: IEEE, 2016. 653–656. [doi: 10.1109/SANER.2016.43]

[36] Tempel S, Brandt T, Lüth C, Dietrich C, Drechsler R. Accurate and extensible symbolic execution of binary code based on formal ISA semantics. In: Proc. of the 2025 Design, Automation & Test in Europe Conf. (DATE). Lyon: IEEE, 2025. 1–7. [doi: 10.23919/DATE64628.2025.10993257]

作者简介

李玉璋, 博士生, CCF 学生会员, 主要研究领域为自动软件工程, 软件供应链安全.

张熙, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为大数据分析, 网络内容安全, 可信人工智能.

徐涛, 博士, 高级工程师, 主要研究领域为计算机系统结构, 系统安全.