软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(10):4623−4641 [doi: 10.13328/j.cnki.jos.006976]

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

反汇编工具中间接跳转表求解算法分析与测试*

庞成宾[1,2], 徐雪兰[1,2], 张天泰[1,2], 茅 兵[1,2]

[1](计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 计算机科学与技术系, 江苏 南京 210023)

通信作者: 茅兵, E-mail: maobing@nju.edu.cn

摘　要: 二进制反汇编是困难的, 但是对于提高二进制软件的安全性至关重要. 造成二进制反汇编比较困难的一大原因是编译器为了提高效率会在二进制代码中引入很多间接跳转表. 为了求解间接跳转表, 主流反汇编工具采用了各种策略. 然而, 这些策略的具体实现以及策略的效果不得而知. 为了帮助研究人员理解反汇编工具的算法实现以及性能, 首先系统总结反汇编工具求解间接跳转表的策略; 然后构建自动化测试间接跳转表框架, 基于该框架, 可以大规模地生成关于间接跳转表的测试集 (包含 2410455 个跳转表); 最后, 在该测试集上, 对反汇编工具求解间接跳转表的性能进行评估, 并人工分析反汇编工具的每个策略引入的错误. 另外, 得益于针对反汇编工具算法实现的系统性总结, 发现 6 个反汇编工具实现上的 bugs.

关键词: 二进制反汇编; 控制流; 间接跳转表; 值集分析

中图法分类号: TP311

中文引用格式: 庞成宾, 徐雪兰, 张天泰, 茅兵. 反汇编工具中间接跳转表求解算法分析与测试. 软件学报, 2024, 35(10): 4623–4641. http://www.jos.org.cn/1000-9825/6976.htm

英文引用格式: Pang CB, Xu XL, Zhang TT, Mao B. Analysis and Testing of Indirect Jump Table Solving Algorithms in Disassembly Tools. Ruan Jian Xue Bao/Journal of Software, 2024, 35(10): 4623–4641 (in Chinese). http://www.jos.org.cn/1000-9825/6976.htm

Analysis and Testing of Indirect Jump Table Solving Algorithms in Disassembly Tools

PANG Cheng-Bin[1,2], XU Xue-Lan[1,2], ZHANG Tian-Tai[1,2], MAO Bing[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: Disassembly of binary codes is hard but necessary for improving the security of binary software. One of the major reasons for the difficult binary disassembly is that the compilers create many indirect jump tables in the binary code for efficiency. In order to solve the targets of the indirect jump table, mainstream disassembly tools use various strategies. However, the details of the implementation of these strategies and their effectiveness are not well studied. To help researchers to well understand the algorithm implementation and performance of disassembly tools, this study first systematically summarizes the strategies used by disassembly tools to solve indirect jump tables; then the study builds an automatic framework for testing indirect jump tables, based on which a large-scale testsuite on indirect jump tables (2410455 jump tables) can be generated. Lastly, this study evaluates the performance of the disassembly tools in solving indirect jump tables on the testsuite and manually analyzes the errors introduced by each strategy of the disassembly tools. In addition, this study finds six bugs in the implementation of the disassembly tools benefiting from the systematic summary of the implementation of the disassembly tool algorithm.

Key words: binary disassembly; control flow; indirect jump table; value set analysis

二进制反汇编是将机器读取的二进制代码转化为方便人类理解的汇编代码的过程. 二进制反汇编是许多安全工作的基础, 比如二进制代码的安全加固[1−7]、二进制代码的相似度检测[8−10]、针对二进制代码打补丁[11]以及二

* 基金项目: 国家自然科学基金 (62032010, 62172201)
收稿时间: 2022-11-30; 修改时间: 2023-02-02, 2023-03-20, 2023-05-08; 采用时间: 2023-05-18; jos 在线出版时间: 2023-10-18
CNKI 网络首发时间: 2023-10-19

进制级别的漏洞挖掘等[12–15]. 然而, 正确地反汇编二进制代码相当困难. 一个主要原因是, 编译器的过度优化引入了间接跳转表等复杂结构, 而间接控制流目标难以通过静态方法确定[16].

在过去的几十年间, 学术界涌现出了众多优秀的反汇编工具, 比如 Angr[17]、Ghidra[18]、radare2[19]和 Dyninst[20]等. 这些工具采用了许多算法以求解间接控制流目标, 包括基于专家知识总结模式进行推断的启发式算法与利用程序分析进行求解的可靠性算法. 然而, 这些算法的细节没有被展示出来, 这些算法求解的结果是否准确或完备也不得而知, 这就给反汇编工具使用者 (比如二进制安全研究人员) 造成困扰. 要消除这一困扰, 我们必须回答以下 3 个问题.

Q1: 现有的反汇编工具采用了哪些启发式与可靠性算法来求解间接跳转表?

Q2: 反汇编工具在恢复间接跳转表的性能 (召回率、准确率) 如何?

Q3: 反汇编工具在恢复间接跳转表时会出现哪些错误? 原因是什么?

为了回答以上 3 个问题, 本文对反汇编工具中恢复间接跳转表所使用的算法进行系统性总结. 研究目标如表 1 所示. 我们从定性与定量两个角度进行研究. 为了回答 Q1, 我们阅读了所有目标工具的源代码以定性地分析其使用的间接跳转表求解算法. 对源代码的阅读使得我们能够准确地掌握这些工具的最新算法, 避免模糊或不明确的描述. 为了回答 Q2 和 Q3, 我们构建了一套框架, 用于自动化测试反汇编工具所使用的间接跳转表求解算法. 该框架首先使用扩展的 Csmith[21]工具生成大量包含 switch-case 语句的 C 文件 (因为 switch-case 语句会被编译器编译为间接跳转表), 然后跟踪主流编译器 (GCC[22]以及 Clang/LLVM[23]) 的编译过程收集间接跳转表作为 Ground Truth[24], 最后将这些 Ground Truth 与反汇编工具的结果进行对比, 对反汇编工具性能进行评估并发现反汇编工具出现的错误. 基于该框架, 我们可以定量地分析反汇编工具恢复间接跳转表的整体性能, 帮助工具使用者对工具性能有更清晰的认知, 进而回答 Q2; 进一步地, 我们也发现了反汇编工具出现的大量错误, 通过人工分析这些错误, 帮助工具开发者改进工具的算法和实现, 进而回答 Q3. 此外, 该框架可以自动地生成跳转表测试用例, 持续性地测试反汇编工具在跳转表求解算法设计或实现上存在的问题.

表 1　目标反汇编工具与基于该工具的代表性工作

工具	版本	出处	基于该工具的工作
Angr	8.19.7.25	Angr官网[17]	文献[25–29]
Ghidra	9.0.4	Ghidra官网[18]	文献[30,31]
radare2	4.4.0	GitHub[19]	文献[32–35]
Dyninst	9.3.3	Dyninst官网[20]	文献[36–40]

通过系统地分析与测试这些工具的间接跳转表算法, 我们有如下发现.

(1) 为了提高结果覆盖率或者工具效率, 主流反汇编工具会采用启发式算法.

(2) 启发式算法一般很难保证结果的正确性.

(3) 反汇编工具可能会采用相似的算法, 但出于对覆盖率与准确率的权衡, 不同的工具会组合不同的算法.

本文的贡献如下.

(1) 本文系统总结了反汇编工具使用的间接跳转表算法, 可以帮助工具使用者对工具实现有更清晰的认知.

(2) 本文扩展了 Csmith 生成算法. 该算法能够自动化生成包含 switch-case 语句的 C 文件.

(3) 本文提出了基于编译器的间接跳转表 Ground Truth 生成算法. 该算法可以在编译过程将 Ground Truth 生成到最终的二进制文件中, 是我们大规模测试反汇编工具的基础.

(4) 本文构建了自动化测试反汇编工具间接跳转表性能的框架. 该框架可以持续地生成测试用例来测试反汇编工具的算法与实现, 进而帮助反汇编工具开发者改进工具.

本文第 1 节介绍本文涉及的基础知识, 以及总结相关研究工作. 第 2 节阐述本文的研究范围. 第 3 节系统性总结 4 个开源反汇编工具恢复间接跳转表所使用的算法. 第 4 节介绍反汇编工具间接跳转表自动化测试的框架. 第

5 节展示反汇编工具的整体性能以及反汇编结果错误分析. 第 6 节对本框架生成的数据集与真实程序构成的数据集进行对比. 第 7 节对本文进行总结.

1 背景与相关工作

本节首先介绍相关背景, 包括间接跳转表形成原因以及 Ground Truth 含义、Csmith 工作原理、值集分析原理. 然后介绍与本文相关的工作.

1.1 间接跳转表及 Ground Truth

为了提升效率, 主流编译器 (比如 GCC、Clang/LLVM 等) 通常会将 switch-case 语句转化为间接跳转表[41]. 在编译器端, 通常由特定的结构 (比如在 LLVM 中是 MachineJumpTableInfo[42]) 存储间接跳转表. 在汇编器端, 这些结构将转化成间接跳转.

如图 1 所示, 编译器会将图 1(a) 的 switch-case 语句编译成图 1(b) 所示的间接跳转表. 间接跳转表如图 1(b) 第 5–9 行所示, 它是包含 N 个表项的地址表, entry1 代表第 1 个 case 的地址. 跳转表的开始地址 (即基地址) 为 base_address, entry_size 表示地址表里每个表项的大小. 为了求解间接跳转表, 反汇编工具需要确定间接跳转处的 base_address、entry_size 以及跳转表的大小 N. 一般来讲, 跳转表中的索引变量 idx 由跳转表的大小 table_size 约束 (如图 1(b) 第 1, 2 行所示).

```
1 switch (index) {
2     case 0;
3     ...;
4     case 1;
5     ...;
6     case n;
7     ...;
8     default: ...;
9 }
```

(a) 源代码

```
1 cmp idx, table_size
2 ja default_case
3 jmp *(.base_address,entry_size,idx)
4   ...
5 .base_address:
6   entry1,
7   entry2,
8   ...,
9   entryN
```

(b) 二进制中对应的间接跳转表

图 1 间接跳转表实例

Ground Truth 是一个相对概念, 是指通过直接观察和测量 (即经验证据) 提供的真实的信息[24]. 间接跳转表 Ground Truth 指的是可以在二进制代码中正确地标记间接跳转表信息.

1.2 Csmith 工作原理

Csmith 是一个随机生成合法 C 程序的工具[21]. 它维护了全局环境与局部环境. 全局环境保存了全局范围的定义: 比如类型、全局变量和函数. 局部环境保存了与当前生成点有关的局部信息, 包括 (1) 当前生成点的函数调用链信息, 该信息被用来做上下文敏感的指针分析; (2) 当前生成点可以引用的变量; (3) 局部变量的别名关系. Csmith 定义了程序代码的生成规则, 支持函数定义、全局变量、局部变量、C 语言表达式、常见控制流 (if/else、函数调用、for、return、break、continue、goto)、数值运算以及位运算等. 为了生成有效的 C 程序, Csmith 首先随机生成 struct 等类型以及全局变量, 然后定义 main 函数按照自上而下的生成规则生成 C 语句: 当定义新的局部变量时, 更新局部环境; 当需要特定类型的变量时, 则从全局或局部环境中选取合适变量, 并更新指针别名关系. 需要注意的是, Csmith 并不支持 switch-case 语句, 因此本文基于 Csmith 生成规则增加了对 switch-case 的支持.

1.3 值集分析

为了求解跳转表的大小, 部分反汇编工具会采用值集 (value set analysis, VSA) 来求解跳转表索引值的范围.本节对值集分析的基础知识进行简单介绍. 值集分析由 Balakrishnan 等人[43]提出. VSA 是一种流敏感、上下文敏感的函数间数据流分析算法. 具体来说, VSA 会将每个内存单元抽象为抽象位置 a-loc (abstract locations). 通过结合数值分析与指针分析算法, VSA 确定每个程序点中每个 a-loc 的数值集与地址集 (称为值集) 的超集. 因此, 同时跟

踪数字值与地址值的数量是 VSA 的关键特点. 这对于二进制文件分析至关重要, 因为数字值和地址值在静态时难以区分[44].

1.4 相关工作

近年来, 为了帮助反汇编工具使用者更好地理解反汇编工具性能以及帮助开发者提高反汇编工具性能, 研究人员对反汇编工具性能进行了大规模评估. 性能评估需要反汇编结果的 Ground Truth. 因此, 接下来将围绕反汇编工具间接跳转表的 Ground Truth 与性能评估、测试集选取进行介绍.

● 反汇编结果 Ground Truth 与性能评估: 为了获得反汇编工具间接跳转表的 Ground Truth, 研究人员使用了不同的方法. 我们将相关方法分为如下几类: (1) 人工构建间接跳转表 Ground Truth. Meng 等人[16]利用专家知识手工构造了几个间接跳转表的 Ground Truth, 该方法虽然能准确获得跳转表 Ground Truth, 但是很难大规模地开展, 这就导致该方法很难准确地反映反汇编工具的性能. (2) 重用已有反汇编工具获得间接跳转表 Ground Truth. Kinder 等人[45]重用反汇编工具 IDA Pro 获取跳转表的 Ground Truth, 该方法虽然简单方便, 但由于反汇编工具实现或算法的问题, 很难获取准确全面的 Ground Truth. (3) 利用编译器的中间输出获得间接跳转表的 Ground Truth. Williams-King 等人[46]利用 GCC 的编译选项-fdump-final-insns 将 GCC 的中间表示输出, 这些中间表示会记录跳转表信息, 比如跳转表的大小. 相应地, 我们也可以使用相似的方式来获得 Clang/LLVM 端的中间输出: 通过编写 LLVM Pass[47]来获得 MachineJumpTableInfo 中存储的间接跳转表信息. 但是这类方法只能获得每个函数存在跳转表的个数和大小, 很难将跳转表信息与间接跳转对应起来. 这就给分析反汇编工具求解间接跳转表失败的原因带来巨大挑战. (4) 利用编译元数据获得间接跳转表 Ground Truth. Andriesse 等人[48]利用编译过程的调试信息获得 Ground Truth. 具体地, 他们首先编写了 LLVM Pass[47]来获得由 LLVM IR 中间语言表示的 switch 语句, 进而 (通过调试信息) 获得每一个 case 语句的行信息, 最后在二进制代码中基于行信息将 case 语句与每个间接跳转表的表项地址对应. 这类方法有以下缺点: 1) 调试信息经过编译器的优化往往是不准确的, 很难精确地对应具体的指令, 导致跳转表 Ground Truth 中的表项地址不正确; 2) 并不是每一个 switch-case 都会被编译器编译成间接跳转表, 这会使得 Ground Truth 出现假阳性. (5) 跟踪编译过程获得跳转表 Ground Truth. 我们的前期工作[49,50]修改了主流编译器 (GCC, Clang/LLVM) 的编译链以在编译过程中获得跳转表 Ground Truth, 并通过污点分析将二进制中的跳转表 Ground Truth 与间接跳转对应起来. 该方法虽然在污点分析的过程中可能存在欠污染 (under-taint) 的问题[50] (我们在实验中并没有发现这种情况), 导致个别间接跳转表 Ground Truth 不能被恢复, 但可以准确地获得间接跳转表 Ground Truth 并将 Ground Truth 与间接跳转对应起来. 接下来, 我们从准确率、召回率、间接跳转与 Ground Truth 匹配度 3 个维度对已有工作进行对比, 如表 2 所示, 圆圈内的填充面积代表 Ground Truth 获取方法在该属性的好坏, 其中填充的面积越多则代表在该属性的表现越好. 从表 2 可以看出, 我们的前期工作[49,50]中跟踪编译过程的方法可以准确地获取并正确对应跳转表 Ground Truth. 因此, 本文采用此类方法来获取间接跳转 Ground Truth, 我们将在第 4.2 节对该方法进行详细介绍.

表 2 间接跳转表 Ground Truth 相关工作对比

Ground Truth 获取方法	相关工作	评估属性		
		准确率	召回率	间接跳转与Ground Truth匹配度
人工构造	[16]	●	○	●
重用已有反汇编工具	[45]	◕	◕	●
利用编译器的中间输出	[46]	●	◕	◔
利用编译元数据	[48]	◕	◕	◔
跟踪编译过程	[49,50]	●	◕	●

相关工作[16,45,46]对反汇编工具的间接跳转表进行了评估, 但是由于 Ground Truth 不准确或者难以规模化, 导致性能评估的结果难以反映反汇编工具的真实性能. 我们前期的工作[49,50]虽然使用了准确的间接跳转表 Ground Truth 获取方法, 但是由于采用了真实应用程序作为测试集, 导致测试集中有很多与间接跳转无关的代码, 这极大

影响测试的效率; 另外, 构建这些数据集需要人的参与, 并且构建测试集效率低、生成的代码包含的跳转表数量少 (详见第 6.1 节). 这就导致很难利用真实程序构建的数据集对反汇编工具的算法和实现进行持续性地测试.

● 测试集选取: 为了评估反汇编工具的性能, 需要构造大规模的测试集. 现有工作[45,46,48,51]大多选取现实生活中使用的开源程序或标准测试集 (比如 Binutils、Coreutils、CPU2006 等), 使用编译器的多个优化等级 (O0、O1、O2、O3 等) 进行手工编译. Meng 等人[16]依靠专家知识手动构造测试集. 与以上依赖人工或专家知识来构造测试集的方法不同, 本文通过扩展 Csmith 来自动化、持续性地生成大量测试集.

2 研究范围

本文研究的是反汇编工具的一个子功能: 间接跳转表的求解. 需要说明的是, 并不是所有的间接跳转都是间接跳转表. 间接跳转一般有两种形态: (1) switch-case 形成的间接跳转表; (2) 经编译器尾调用优化[52]之后的间接函数调用. 出于以下原因, 我们选择间接跳转表作为研究目标: (1) 大部分反汇编工具只针对间接跳转表进行特殊处理, 比如 Dyninst、Ghidra、radare2 都需要判定当前间接跳转是否为跳转表; (2) 间接函数调用目标的 Ground Truth 很难通过静态获取.

考虑到大部分相关工作[16,45,46]针对的体系结构是 x86/x64 平台, 本文也只考虑 x86/x64 体系结构. 我们选取了如前文表 1 所示的主流开源反汇编工具. 选取标准如下: (1) 由于需要剖析反汇编工具的内部实现, 工具必须是开源的, 在此标准的约束下, 不考虑 IDA Pro、Binary Ninja 这些没有源代码的商业软件; (2) 研究的是间接跳转表这一子功能, 因此也不考虑 BAP、ByteWeight、Objdump 这类没有间接跳转表求解的反汇编工具. (3) 研究的反汇编工具支持大规模自动化运行, 否则很难针对这些工具进行系统评估与测试. 因此, 我们排除 JakStab, 该工具在解析二进制文件出现问题 (https://github.com/jkinder/jakstab/issues/9).

3 间接跳转表求解算法分析

为了深入了解反汇编工具求解间接跳转表的内部算法, 对前文表 1 所示的开源反汇编工具的代码进行分析. 接下来, 我们首先具体介绍每种工具的实现细节, 然后对工具使用的算法进行归纳总结.

radare2 总结了 4 种模式来求解间接跳转表, 这些模式如图 2 所示. 具体算法如下: (1) 当遇到间接跳转时, 它会首先判断该跳转是否为间接跳转表. 如类型 1 所示, 如果该间接跳转形式是 jmpq *(base, %r2, size), 则判定为跳转表, 并可以确定跳转表的基地址; 否则, 则需要线性扫描间接跳转之前的指令, 如果找到形似类型 2, 3, 4 中展示的 mov base(, %r2, size), %r3、mov (%r2, %r3, size), %r4 或 mov (%r1, %r3, size), %r4 指令, 则认为该间接跳转为间接跳转表, 并找到相应的跳转表基地址. 需要注意的是, 在类型 3 中, radare2 限制了扫描长度为 60 字节; (2) 为了确定跳转表的大小, radare2 会根据控制流找到间接跳转所在基本块前面的父基本块, 并检查该基本块是否为条件跳转, 接着判断条件变量是否由 cmp/sub 指令确定, 最后判断该 cmp/sub 指令是否包含一个常数操作数 (如类型 1, 2, 3, 4 所示). 如果以上条件都满足, 则将 cmp/sub 指令中的常数操作数作为跳转表的大小. 此外, radare2 会过滤掉大小大于 512 的跳转表. radare2 使用了大量的专家知识来求解间接跳转表, 由于不同编译器不同优化等级产生的跳转表形式各异, 导致 radare2 的效果比较差 (如第 5 节所示).

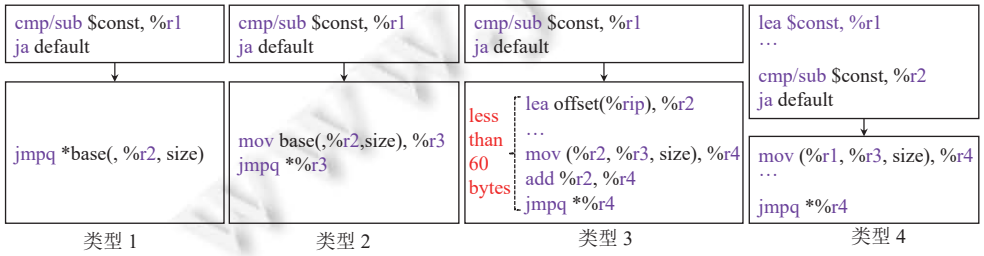


图 2 radare2 可以处理的间接跳转表的模式

Dyninst 的跳转表求解算法如图 3(a) 所示, 上半部分展示的是一个间接跳转表的二进制代码, 下半部分展示 Dyninst 的求解过程. 算法由两个过程组成: (1) 首先, Dyninst 从间接跳转指令开始进行后向切片 (program slicing)[53]. 每当遇到一个新的二进制指令时, 首先判断当前指令是否包含内存访问操作, 然后判断此访问的内存是否可以简化为 [r9 + r8 * const] 形式, 是则判定此间接跳转为跳转表并确定跳转表基地址为 r9, 跳转表每个表项大小为 const 表示的大小; (2) 当确定了间接跳转表形式后, Dyninst 继续后向切片, 在切片过程中会对跳转表大小进行 VSA 分析, 如果能确定表的大小, 则停止切片. 此外, 为了效率, Dyninst 会限制切片大小为 50 个语句.

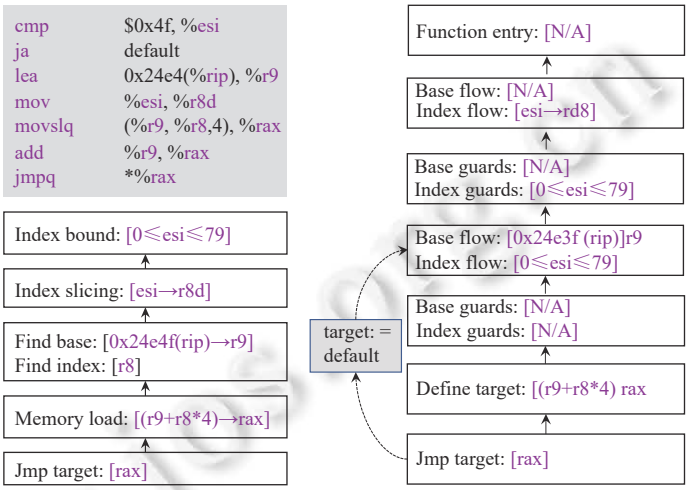


图 3 Dyninst 和 Ghidra 间接跳转表求解算法

Ghidra 的跳转表求解算法如图 3(b) 所示. 首先, Ghidra 会将间接跳转假设为跳转表, 然后对当前跳转表做如下分析: 在当前函数中, Ghidra 会寻找一条同时定义基地址和索引的单一路径(如图中实线所指路径), 并识别出间接跳转表的 default 分支 (指在跳转表范围外的分支, 对应于 switch 里面的 default 语句, 如图 3 中虚线所指分支, 其中 target:=default 代表的是在该分支中目标地址是 default). 接着, 在实线所示路径上跟踪基地值与索引的传播过程, 先确定基地址与索引的表示形式 (在此处分别为 r9 与 r8), 再利用 VSA 来求解它们的范围. 在该例子中, Base guards 和 Index guards 分别表示求解过程中基地址与索引的范围, [N/A] 代表暂时没有求解出具体范围或值; Base flow 和 Index flow 分别代表基地址与索引的传播过程. 为了提高精度, Ghidra 会限制跳转表大小不超过 1 024, 并过滤掉超过 1 024 的跳转表.

Angr 同样使用 VSA 对间接跳转表目标进行求解. 具体来说, 当遇到间接跳转时, Angr 会从间接跳转表所在的基本块往前回溯 3 个基本块, 并从这些基本块开始进行 VSA 求解. 与 Ghidra 类似, 为了提高精度, Angr 会筛选求解的跳转表的大小: 如果跳转表大小超过 1 000 000, 则认为该结果非法, 将该结果过滤.

如表 3 所示, 我们从可靠性算法与启发式算法两个角度将以上描述的算法进行系统总结.

表 3 开源反汇编工具间接跳转表求解算法总结

类型	算法描述	工具
可靠性算法	使用VSA求解跳转表目标	Angr、Ghidra、Dyninst
启发式算法	基于专家知识确定间接跳转表	Ghidra、Dyninst、radare2
	限制VSA时切片的长度	Angr、Dyninst
	过滤掉跳转表大小超过阈值的	Angr、Ghidra、radare2

从表 3 可以看出, 这 4 个反汇编工具都采用了启发式的算法对求解的结果进行限制. 具体地, Angr、Ghidra、radare2 都会过滤掉大小超过阈值的跳转表. 这一算法虽然能够提高反汇编工具的精度, 但也会因此漏掉一些正确的跳转表. 我们通过对前期工作[49,50]中构建的 x86/x64 数据集分析发现, 大小大于 512 和 1 024 的间接跳转表分别为 2 435 与 51 个. 这表明该启发式算法会过滤掉一些正确的跳转表. 同时, Angr、Dyninst 会限制 VSA 过程中切片的长度以提高求解效率, 然而根据第 5.4 节分析可知, 分别有 84.6% (Angr) 和 9.4% (Dyninst) 的漏报是由该启发式算法导致的.

4 自动化测试间接跳转表技术

为了自动化评估反汇编工具性能, 本文提出了一种自动化测试间接跳转表技术. 流程如图 4 所示: (1) 首先, 将 Csmith 扩展到 SCsmith, 该工具能自动化生成包含 switch-case 语句的 C 文件; (2) 接着, 将 C 文件交给 OracleGT 编译器 (Oracle Ground Truth, 通过跟踪主流编译器的编译过程来自动化地收集间接跳转表信息, 并将其作为 Ground Truth 信息生成到最终的二进制文件中) 以不同优化等级进行编译 (比如 O0、O1、O2、O3、Os、Ofast 等). 其中图 4 中的 MC (machine code) Layer 指的是 LLVM 中用于表示和生成机器码的阶段[54], GAS (GNU assembler) 指的是 Binutils[55]中的汇编器; (3) 至此, 我们自动化构建了间接跳转表的测试集; (4) 然后, 将测试集中的二进制文件交给被测试的反汇编工具, 获得这些工具的间接跳转表结果; (5) 最后, 通过对比 Ground Truth 与反汇编工具的结果, 可以对反汇编工具的性能 (覆盖率、准确度) 进行评估, 对反汇编工具的算法进行持续性地测试, 进一步人工分析反汇编工具的错误 bug. 接下来, 我们会详细介绍 SCsmith 与 OracleGT 的实现细节.

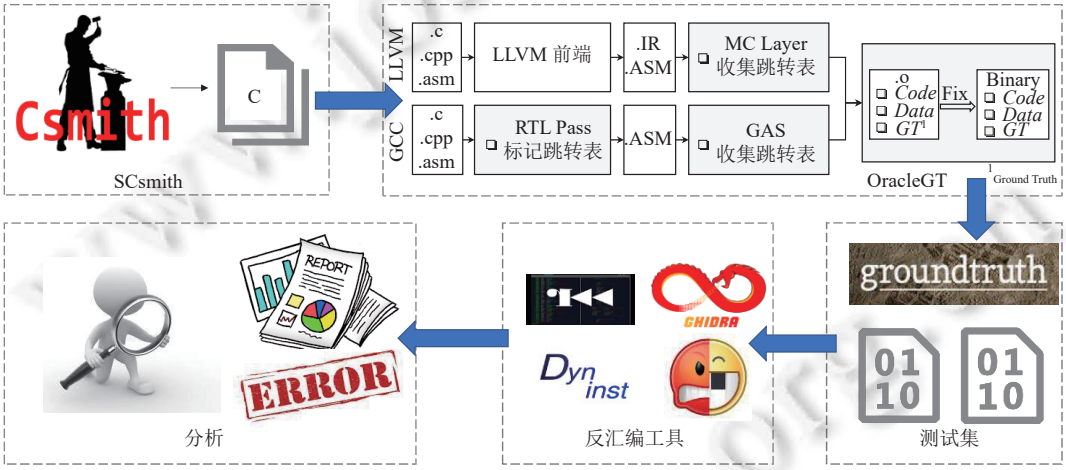


图 4 自动化测试间接跳转表框架

4.1 SCsmith

本文设计了 SCsmith 工具. 该工具对 Csmith-2.3.0 进行扩展, 使其可以在 C 文件中生成 switch-case 语句. 如第 1.2 节所述, Csmith 定义了自上而下的生成规则. 根据规则, Csmith 生成若干函数, 每个函数包含若干局部变量与语句 (statements), 其中每个语句既可以是简单的数值运算, 也可以是 for、if 等控制流结构. 对每个函数, 我们扩展如下规则来生成 switch-case 语句.

(1) 假设当前函数中包含 T 个语句, SCsmith 会随机在第 i ($0 \leqslant i < T$) 个语句前找到一个插入点插入 switch 语句, 并随机选取当前可用的局部或全局数值类型 (char、short、int、long 或 enum) 变量作为 switch 语句的表达式. 此时当前 switch 可包含的语句个数为 $m = T - i$.

(2) 接着 SCsmith 会随机确定当前“跳转表”的大小为 N, N 决定了 case 语句中常量表达式的上限. 为了使 switch-case 语句更有机会被编译器编译成跳转表, SCsmith 定义 N 的范围为 $2 \leqslant N < 2 \times m$.

(3) 最后, SCsmith 在 $[i,T)$ 语句之间随机插入 c 个 case 语句. 为了保证每个 case 至少包含一个语句, c 满足 $2 \leqslant c < \min(N,m)$. 为了保证语义正确, SCsmith 还需保证 case 之间的常量表达式互不相同, 且有一个常量表达式为 $N-1$ (这决定了编译出的跳转表的大小为 N).

接下来, 我们通过一个例子来展示 SCsmith 的工作过程. 图 5(a) 展示的是 Csmith 生成的原程序中的一个函数, 在该函数开始处, 会首先进行局部变量的初始化 (如第 3 行所示). 接下来的几行是对已初始化的局部变量或函数参数进行引用的语句 (需要说明的是 for 循环及里面的操作也被 SCsmith 视作一个语句). 图 5(b) 展示的是 SCsmith 修改后的函数, 其中用红色标识的语句是由 SCsmith 插入的. 具体地, SCsmith 会首先选择一个插入点 (在该例子中插入点是图 5(a) 中第 5 行), 然后搜索在插入点处可用的数值类型的变量 (比如 a, b, c, i 与*p, 在该例子中选择了*p), 将选中的变量 (*p) 作为 switch 语句的表达式 (如图 5(b) 中第 5 行所示). 此时, SCsmith 可以计算出该 switch 可以包含的语句个数为 4 (图 5(a) 中第 5–10 行的语句, 其中整个 for 循环被视作 1 个语句). 接着, SCsmith 随机确定了跳转表的大小为 N=6. 最后, SCsmith 在接下来的语句中随机插入 4 个 case 语句, 并在每个 case 下面随机确定是否添加 break 语句. 需要说明的是, 由于新增的 switch/case 包含的语句只会涉及对变量的引用, 并没有定义新的变量, 也就不存在变量的生命周期的变化. 因此, SCsmith 不会引入新的编译错误.

```
1  void func(int* p) {
2  // 局部变量初始化
3  int a = 1, b = 2, c = 3, i = 0;
4  ...
5  *p += a;
6  c += 2;
7  b = *p;
8  for (i = 0; i < c; i++) {
9  ...
10 }
11 }
```

(a) Csmith 生成的函数

```
1  void func(int* p) {
2  // 局部变量初始化
3  int a = 1, b = 2, c = 3, i = 0;
4  ...
5  switch(*p) {
6  case 0:
7      *p += a; break;
8  case 2:
9      c += 2; break;
10 case 3:
11     b = *p;
12 case 5:
13     for (i = 0; i < c; i++) {
14     ...}; break;} }
```

(b) SCsmith 插入 switch/case 后的函数

图 5 SCsmith 插入 switch/case 语句实例

4.2 OracleGT

为了效率, 编译器通常会将 switch-case 转换成间接跳转表形式. 具体来说, 首先, 编译器将 switch 转为相应的中间语言表示 (LLVM 中使用 switch IR 表示, GCC 中使用 GIMPILE_SWITCH 表示). 然后, 编译器会将 switch 中间表示转换为 Jump Table 结构 (LLVM 中使用 MachineJumpTableInfo 结构表示, GCC 中使用 rtx_jump_table_data 表示), 该结构记录了跳转表大小以及跳转表中每个表项的目标地址. 接着, 编译器会将跳转表数据生成到汇编代码或者二进制文件中. 因此, 通过跟踪编译过程, 本文可以获得跳转表 Ground Truth, 具体流程如前文图 4 右上角 OracleGT 所示. 接下来将以 GCC 编译器为例, 详细介绍 OracleGT 如何在编译器、汇编器与链接器中收集跳转表 Ground Truth. OracleGT 现在支持主流编译器 GCC 的 8.1 版本与 LLVM 的 6.0 版本.

● 编译器: 当 GCC 后端完成了 RTL (register transfer language)[56]中间表示的最后一个 Pass 时, 它会将 RTL 中间语言转换为汇编语言. GCC 会遍历每一条 RTL 中间表示来将其代表的指令/数据生成到汇编文件, 在这个过程中, OracleGT 会劫持 GCC 将 rtx_jmp_table_data 转为汇编代码中跳转表的过程, 并定义额外的汇编伪指令 (assembler directives)[57]来标记跳转表信息 (OracleGT 在 GCC 端插入的代码如图 6 所示, 其中第 9–16 行为 OracleGT 插入的代码). 以图 7 为例, 第 6–8 行之间是生成的跳转表. 我们定义了.bbinfo_JMPTBL 来记录跳转表信息, 该伪指令后面有两个常数操作数, 分别代表跳转表大小以及跳转表中每个表项的大小. 在该例子中, .L1001 标签之后的 11 个单元是跳转表中的所有目标, 每个目标的大小是 8 字节.

● 汇编器: 汇编器会线性地解析汇编文件中的所有指令. 当遇到.bbinfo_JMPTBL 伪指令时, 我们会收集当前

跳转表的信息, 并记录下一个标签 (如图 7 中的.L1001) 为跳转表的基地址, 并将该跳转表信息与基地址的引用相关联 (如图 7 中将第 3 行中.L1001 的引用与跳转表信息关联). 接着, OracleGT 会记录基地址引用在目标文件中的偏移.

```
1  // Iterates over every RTL instruction and emits it into the assembly file
2  static rtx_insn *
3  final_scan_insn_1 (rtx_insn *insn, FILE *file, int optimize_p ATTRIBUTE_UNUSED,
4                     int nopeepholes ATTRIBUTE_UNUSED, int *seen) {
5       ...
6      // Detects that the insn is jump table
7     if (JUMP_TABLE_DATA_P (insn))
8     {
9          rtx_jump_table_data *table_tmp = dyn_cast<rtx_jump_table_data*>(insn);
10         // get the size of the jump table
11         vlen = XVECLEN (body, GET_CODE (body) == ADDR_DIFF_VEC);
12         // get the size of every table entry
13         uint32_t entry_size = GET_MODE_SIZE(table_tmp->get_data_mode());
14         // dump .bbInfo_JMPTBL table_size entry_size
15         fprintf(asm_out_file, "\t.%s %d %d\n", \
16            "bbInfo_JMPTBL", table_size, entry_size);
17         ...
18         // dump contents of jump table
19    }
20 }
```

图 6　OracleGT 劫持 GCC 端以在汇编文件中插入跳转表 Ground Truth 的相关代码

```
1 cmpl   $10, %r8d          5 .bbInfo_JMPTBL 11 8
2 ja     .L1021             6 .L1001
3 mov    rax, .L1001(,%r8,8) 7 .quad .L1003
4 jmp    *rax               8 ...
```

图 7　OracleGT 在汇编代码标记跳转表信息实例

● 链接器: 在该阶段, 链接器会合并不同的目标文件以生成最终的二进制文件. OracleGT 在该过程中会更新跳转表在最终二进制文件中的偏移.

然而, 经过链接后获取的跳转表 Ground Truth 只能与跳转表的交叉引用 (cross reference, 在此指的是对跳转表基地址的引用) 对应起来, 并不能对应到最终的间接跳转指令. 如在图 7 的例子中, 我们只能将跳转表 Ground Truth 与第 3 行中的交叉引用.L1001 对应起来, 而不能直接对应到第 4 行的间接跳转. 为了对反汇编工具进行更精确的评估以及方便研究人员分析跳转表错误, OracleGT 需要在链接后将跳转表信息与相应的间接跳转进行关联. 具体来说, OracleGT 从指向跳转表基地址的引用开始进行污点分析 (taint analysis) 以找到与其对应的间接跳转. 我们的污点分析实现如算法 1 所示. 我们在控制流图的每个指令处跟踪并更新寄存器与栈内存的污点信息. 具体地, 首先会对包含跳转表交叉引用的指令进行污点信息的初始化 (算法 1 中第 22 行的 taint_initialize 函数): 将该指令的所有被用来写入的寄存器和栈内存染色. 然后根据控制流信息来更新下一条指令中寄存器与栈内存的污点信息: 首先需要判断该指令的源操作数是否被染色 (如算法 1 中 4–8 行所示, 我们遍历所有源操作数中的寄存器与栈内存), 如果源操作数被染色, 需要将目标操作数中的寄存器与栈内存染色; 否则, 我们需要清除目标操作数中寄存器与栈内存的染色信息 (如算法 1 中 9–15 行所示). 如算法 1 中 27–33 行所示, 如果当前指令是间接跳转并且源操作数中寄存器或栈内存被染色 ($\mathcal{I}$.is_tainted() 函数的操作), 则找到了相应的间接跳转指令. 此时, 会将间接跳转的目标地址更新到控制流图上, 以方便其他跳转表交叉引用污点分析的执行. 需要说明的是, 针对栈内存的污点信息, 我们目前只能跟踪直接的栈内存访问 (比如通过 rbp, rsp 寻址的栈内存). 如果一些栈内存通过内存别名进行访问, 我们会丢失相应栈内存的污点信息 (欠污染, under-taint). 但我们的实验发现, 针对跳转表交叉引用的污点信息传递都发生在寄存器或者直接访问的栈内存, 并没有发现 under-taint 的情况.

算法 1. 间接跳转目标查找算法.

输入: 识别出的间接跳转表对应的交叉引用数组: $\overrightarrow{JTR}=\{jtr_1, jtr_2, \ldots, jtr_n\}$; 控制流图: CFG;

输出: 间接跳转表对应的交叉引用与跳转表映射数组: $\overrightarrow{M}=\{(jtr_1, ij_1), (jtr_2, ij_2), \ldots, (jtr_n, ij_n)\}$. /* ij_i 代表找到的间接跳转 */

```
1 初始化: M⃗ = ∅; fixpoint = false;
2 Procedure taint_instruction(𝓘):
3         tainted = false
4         for each register ℛ_r or stack memory 𝒮_r used for reading in 𝓘 do
5           if ℛ_r.is_tainted() or 𝒮_r.is_tainted() then
6                 tainted = true
7           end
8         end
9         for each register ℛ_w or stack memory 𝒮_w used for writing in 𝓘 do
10            if tainted then
11              ℛ_w.taint() || 𝒮_w.taint()
12            then
13              ℛ_w.clear_taint() || 𝒮_w.clear_taint()
14            end
15        end
16 return
17 while ¬fixpoint do /* 一直循环, 直到 CFG 没有更新 */
18    fixpoint = true
19    for each jtr_i in JTR⃗ do
20       Q⃗ = ∅
21       𝓘 = CFG.get_instr(jtr_i) /* 获取包含 jtr_i 的指令 */
22       𝓘.taint_initialize()
23       Q⃗.push(𝓘)
24       while Q⃗.is_not_empty() do
25          𝓘 = Q⃗.pop()
26          taint_instruction(𝓘)
27          if 𝓘.is_tainted() and 𝓘.is_indirect_jump() then
28             fixpoint = false
29             M⃗.add((jtr_i, instruction))
30             CFG.update(ℐ, jtr_i) /* 根据求解的跳转表信息更新 CFG */
31             JTR⃗.remove(jtr_i)
32             break
33          end
34          Q⃗.append(CFG.get_successors(𝓘))
35       end
36    end
```

37 **end**
38 **return** $\vec{M}$

5 性能评估与错误分析

为了验证自动化测试间接跳转框架的有效性以及评估反汇编工具的性能, 本节从以下 3 个方面进行介绍: (1) 展示生成的测试集数据; (2) 通过计算召回率和准确率来评估反汇编工具性能; (3) 结合反汇编算法手动分析反汇编工具性能.

5.1 实验环境

本文实验在 Intel i7-10700K CPU 3.80 GHz, Ubuntu 20.04 操作系统上进行. OracleGT 支持 GCC-8.1 与 Clang/LLVM-6.0, 使用的汇编器和链接器分别是 Gas-2.30 与 Gold-2.30. SCsmith 基于 Csmith-2.3.0 版本修改. 我们在编译器端增加了约 1 100 行 C/C++代码以支持 Ground Truth 的生成, 在 Csmith 上增加了约 200 行 C/C++代码以生成 switch-case 语句, 我们写了约 1 200 行 Python/bash 代码以提取 Ground Truth 和比较反汇编工具的结果.

5.2 测试集数据

为了生成包含 switch-case 语句的 C 文件, 我们在实验环境中运行了 10 个 SCsmith 生成实例 (每个实例占用一个 CPU 进程). 在为期 2 天的生成中, 我们共获得 111 085 个 C 文件. 经编译器验证, 其中 110 929 个 C 文件可以被正确编译成二进制文件, 正确率在 99.99% 以上. 接着, 我们使用 OracleGT 中的 GCC-8.1 与 Clang/LLVM-6.0 编译器分别将这些 C 文件以不同的优化等级 (O0、O1、O2、O3、Os、Ofast) 自动化编译成了 x86 与 x64 两个体系架构下共计 1 331 148 个可执行二进制文件 (构建这些二进制文件需要 31 h), 并统计这些二进制文件中的间接跳转表数量, 结果如表 4 所示, 共包含 2 410 455 个间接跳转表. 我们发现, 在不同优化等级下, 编译器生成的跳转表个数不尽相同, 高优化等级下 (如 O2、O3) 生成的跳转表个数较少. 这是因为编译器并不会将所有的 switch/case 转化为间接跳转表. 编译器至少做两种检查: 1) 检查 switch 语句表达式的值范围和所有 case 值的范围是否有交集, 如果确认没有交集, 则将该 switch 删除; 2) 检查 switch 语句所在的基本块是否为死代码, 如果确认该基本块是死代码, 则将基本块和 switch 删除. 在较高的优化等级下, 这些检查更准确, 因此在高优化等级下会删除更多无用的 switch/case. 我们还发现, x86 与 x64 架构下的跳转表数量有差异. 这是因为编译器采用静态数据流分析求解 switch 语句表达式的值范围, 而不同平台下指针大小不同, 致使 x86 和 x64 下的值范围求解结果可能不相同, 这就导致不同平台下第 1 种检查的结果不一致. 总的来说, 表 4 的结果证明 SCsmith 能在短时间内生成大量包含 switch-case 的 C 文件, 这些 C 文件能被正确编译成二进制, 并包含大量的间接跳转表. 在此基础上, 我们可以自动化构建间接跳转表测试集.

表 4 间接跳转表个数统计

架构	O0	O1	O2	O3	Os	Ofast
x86	762 811	100 968	73 004	81 152	70 052	81 138
x64	803 826	113 759	78 865	81 749	81 197	81 934
总计	1 566 637	214 727	151 869	162 901	151 249	163 072

5.3 反汇编工具性能评估

基于第 5.2 节构建的测试集, 我们对反汇编工具恢复间接跳转表的准确率 (Precision) 与召回率 (Recall) 进行系统性评估. 图 8 分别展示了 x86 与 x64 架构下不同优化等级下的结果.

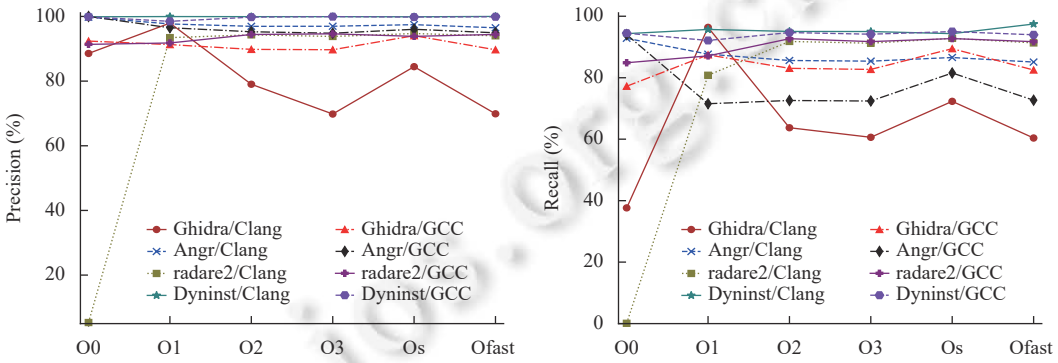
根据图 8 的结果, 我们有如下发现.

(1) 反汇编工具在不同优化等级下性能各异. Angr 在高优化等级下的表现较差, 因为 Angr 为追求效率会限制

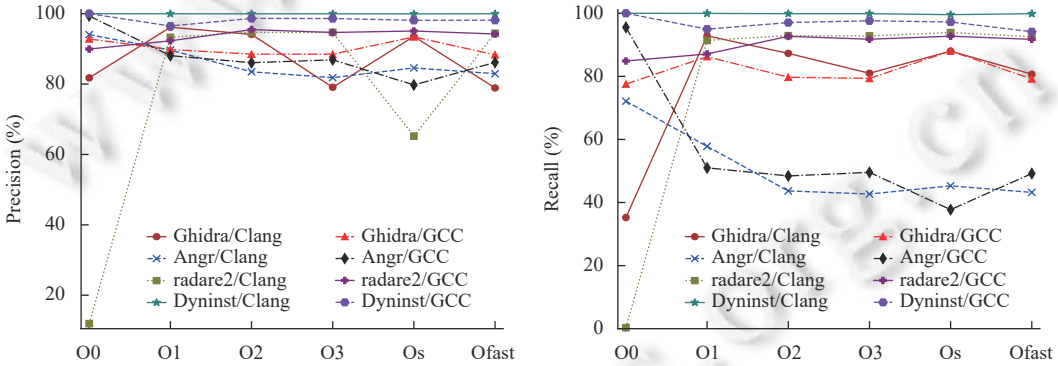
切片范围为 3 个基本块, 而编译器的优化会改变代码的布局, 导致 Angr 的切片范围不一定能覆盖关于跳转表索引的限制. 总体上来看, Dyninst 表现最好, 这有两方面原因: 1) Dyninst 使用切片对与间接跳转有关的内存访问进行合并简化, 然后判断简化后的内存访问是否符合跳转表模式, 这比 radare2 基于专家知识人工总结的 4 个跳转表模式更具一般化, 使得 Dyninst 可以适应不同的优化等级. 2) Dyninst 内部实现的 VSA 支持的指令更完整. 相对于 Angr 无法支持 sbb 指令, Ghidra 没有考虑 sub 指令来说, Dyninst 几乎支持了常见的 x86/x64 指令.

(2) 反汇编工具在不同编译器下的表现有差异. 比如, radare2 在 Clang O0 的表现比较差, 因为 radare2 依靠专家知识总结的模式不完整, 不能很好地解决 Clang O0 编译的跳转表.

(3) 反汇编工具在不同体系结构下的表现也有差异. 比如, Angr 在 x86 的召回率为 82.2%, 而在 x64 的召回率仅为 53.1%.



(a) x86 架构



(b) x64 架构

图 8 反汇编工具在 x86/x64 架构下多个优化等级与编译器性能对比

5.4 反汇编工具错误分析

为了分析反汇编工具出错原因, 我们对每个工具分别随机选取 10% 的假阳性与假阴性例子, 手动分析错误原因. 以下百分数为在该工具中随机抽取的例子中错误原因的比例. 结果如表 5 和表 6 所示.

(1) 据第 3 节的描述, radare2、Ghidra 和 Dyninst 都采用基于专家知识的启发式算法, 这导致了一些假阴性. 在随机抽取的假阴性例子中, 其中 radare2 有 100% 是由基于专家知识的启发式算法引入的, Ghidra 和 Dyninst 的比例分别为 32.4% 和 90.1%.

(2) Ghidra 假阴性: 在 VSA 求解的过程中没有考虑 sub 指令, 这导致了 61.3% 的假阴性; 另外, Ghidra 在实现过程中没有正确跟踪针对跳转表索引的限制条件, 这导致了 6.3% 的假阴性.

表 5 反汇编工具假阴性错误率统计 (%)

工具	启发式算法	错误率
Ghidra	专家知识引入	32.4
	VSA没有考虑sub指令	61.3
	跳转表索引限制条件实现存在问题	6.3
Dyninst	专家知识引入	90.1
	切片长度限制	9.4
	间接跳转隐式限制	0.5
Angr	切片长度限制	84.6
	VSA无法正确处理sbb指令	15.4

表 6 反汇编工具假阳性错误率统计 (%)

工具	启发式算法	错误率
Ghidra	间接跳转表转为间接函数调用	56.4
	VSA没有考虑sub指令	33.2
	收集的索引变量限制条件不准确	4.5
	将跳转表的默认分支作为跳转目标	5.9
Dyninst	无法正确处理索引变量别名	79.4
	没有考虑变量类型本身的范围限制	16.8
	切片长度限制	3.8
Angr	切片长度限制	78.6
	切片过程中忽略了某些路径	21.4

(3) Dyninst 假阴性: 由于限制切片长度, 导致了 9.4% 的假阴性; 我们还发现少量例子是由于没有对间接跳转表索引变量的范围进行显式限制 (如图 9 所示, 在二进制代码中没有针对索引变量 which 的显式限制), 这导致了 0.5% 的假阴性.

```
switch(which){
    case 't': ...
    ...
    default:
        undefined();
}
```

```
; no restriction on %al
sub    $0x64,%eax
movzbl %al,%eax
movslq (%r10,%rax,4),%rax
add    %r10,%rax
jmpq   *%rax
```

图 9 对跳转表索引没有显式的实例

(4) Angr 假阴性: 由于限制切片长度, VSA 无法求解结果, 这导致了 84.6% 的假阴性; Angr 的 VSA 实现无法正确处理 sbb 指令, 这又造成了 15.4% 的假阴性.

(5) radare2 假阳性: 据第 3 节的描述, radare2 通过匹配 sub/cmp 指令来确定跳转表大小, 然而该策略为其带来了 100% 的假阳性.

(6) Ghidra 假阳性: 在处理不了间接跳转时, Ghidra 会将间接跳转表转为间接函数调用, 并使用常量传播算法求解部分目标, 这一做法引入了 56.4% 的假阳性; 由于在 VSA 实现时没有考虑 sub 指令, 引入了 33.2% 的假阳性; 在收集针对索引变量的限制条件时, 会出现不准确或错误的情况, 这导致了 4.5% 的假阳性; 另外, Ghidra 会将一些跳转表的默认分支作为另一个跳转目标, 这也导致了 5.9% 的假阳性.

(7) Dyninst 假阳性: 由于无法正确处理索引变量的别名问题, 导致了 79.4% 的假阳性; 由于没有考虑变量类型本身的范围限制 (比如 1 字节的变量的范围在 0–255 之间), 导致了 16.8% 的假阳性; 此外, 限制了切片大小导致了 3.8% 的假阳性.

(8) Angr 假阳性: 由于切片长度有限导致了 78.6% 的假阳性; 在切片过程中忽略了某些路径, 导致了 21.4% 的假阳性.

此外, 我们还发现了 6 个反汇编工具在处理跳转表实现上的 bugs. 我们将这些 bugs 上报给了开发者并提供了相应的解决方案, 得到了开发者们积极地回应. bugs 介绍如表 7 所示.

接下来, 我们通过展示第 6 个 bug 来说明本框架找反汇编工具跳转表求解的有效性. 该 bug 是 Ghidra 在对跳转表索引值进行值集分析 (VSA) 时出现的逻辑性错误. 具体来说, Ghidra 定义了 CircleRange 类来表示索引值的取值范围, 并根据不同的指令来更新索引值的范围. 需要说明的是, Ghidra 定义了 P-Code[58]中间表示指令来统一 VSA 求解过程. Ghidra 首先将汇编指令 lift 到 P-Code 指令, 然后在 P-Code 指令上进行 VSA 求解, 这么做的好处是简化了 VSA 过程中模拟指令语义的过程. CircleRange 通过定义如下形式来表示跳转表索引值 *index* 的取值范围.

$$index = [left, step, right),$$

其中, $left$ 和 $right$ 分别定义了 $index$ 取值范围的左边界与右边界 (右边界的值取不到, 即 $left \ll index < right$), $step$ 定义了 $index$ 在该范围内的取值间隔. 例如, $index = [-2,2,2)$, 则 $index$ 的值为−2 和 0.

表 7 反汇编工具实现上的 bugs

No.	工具	错误描述	相关链接
1	Angr	在VSA过程中无法正确处理SBB指令	https://github.com/Angr/Angr/issues/2039
2	Angr	无法正确处理ud2指令	https://github.com/Angr/Angr/issues/1963
3	Dyninst	在VSA过程中无法正确处理rip相关地址	https://github.com/Dyninst/Dyninst/issues/694
4	Dyninst	在VSA过程中错误地假设了所有全局变量是只读变量	https://github.com/Dyninst/Dyninst/issues/692
5	Dyninst	无法正确处理adcx指令	https://github.com/Dyninst/Dyninst/issues/766
6	Ghidra	在VSA过程中对CPUI_INT_NEGATE指令 (Ghidra的中间指令)的处理有逻辑错误	https://github.com/NationalSecurityAgency/Ghidra/issues/3064

P-Code 中定义了 INT_NEGATE[59]指令. 该指令有一个输入参数, 它的逻辑是对输入参数进行按位取反操作, 即 $output = \sim input$. 然而, Ghidra 在 VSA 过程中对 INT_NEGATE 进行模拟执行时, 错误地将按位取反的逻辑解释为乘以−1 的操作, 导致了结果错误, 具体 bug 如图 10(a) 所示. 比如, 若变量 a 的取值范围为 $[-2,2,2)$, 则 INT_NEGATE(a) 的值应该是 $[-1,2,3)$, 而不是 $[-2,2,2)$. 我们给出了相应的修正版本 (图 10(b) 所示) 并将该补丁提交给开发者, 在 3 天内得到了 Ghidra 开发者的确认, 该版本已合并到下一个版本中. 这个 bug 说明, 我们的工作不仅可以系统性评估反汇编工具的性能, 还能发现反汇编工具在设计或实现上存在的问题, 并结合人工分析将这些问题定位出来.

```
# buggy code
step = input.step;
mask = input.mask;
left = -input.right & mask;
right = -input.right & mask
```

(a) 出现 bug 的版本

```
# our patched code
step = input.step;
mask = input.mask;
left = (-input.right + step) & mask;
right = (-input.left + step) & mask;
```

(b) 提交补丁的版本

图 10 Ghidra 处理 INT_NEGATE 逻辑

6 对比讨论

我们的前期工作[49,50]使用人工构建的由真实程序构成的数据集进行反汇编工具的性能评估. 与之相比, 我们使用 SCsmith 生成的数据集, 这有 2 个优势: (1) 高效率、自动化、有效地生成大型数据集; (2) 在 (1) 的基础上, 提供持续性测试的机会: 作为真实程序数据集的补充, 发现更多反汇编工具算法和实现的错误. 我们将在第 6.1 节讨论优势 (1), 在第 6.2 节讨论优势 (2).

6.1 数据集

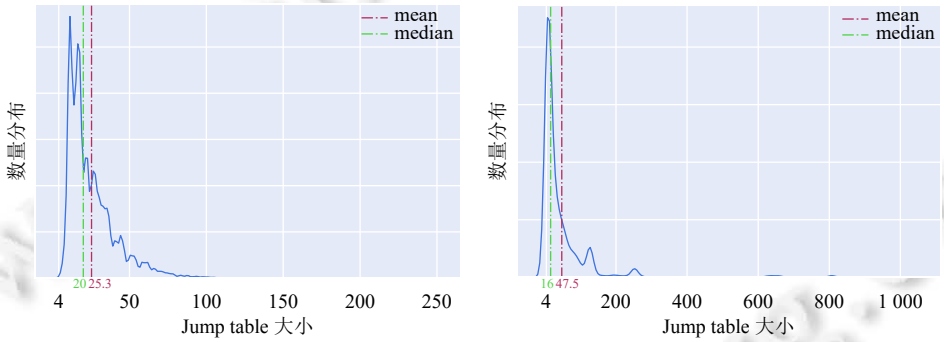
我们将从跳转表个数、生成效率、构建过程是否自动化以及密度进行多方面对比. 首先, 我们对前期工作[49,50]中构建的真实程序数据集的跳转表个数进行统计. 该数据集共有 3 985 个 Linux 二进制文件, 共包含 105 424 个间接跳转表, 远远小于 SCsmith 生成的 1 331 148 个二进制文件中包含的 2 410 455 个跳转表. 其次, 前期工作[49,50]从收集这些真实程序到构建不同优化等级/编译器/架构下的二进制, 一共需要至少 336 h, 而 SCsmith 生成与编译的效率更高效 (生成 111 085 个 C 文件需要 2 天, 再将这些 C 文件构建成多个优化等级/编译器/架构下的二进制也只需要 31 h). 同时, 构建真实程序数据集的大部分过程需要人工参与, 包括收集下载真实程序的源代码与配置编译环境, 相比来看, 本文提供的框架从生成 C 文件到编译成二进制都是自动化的过程. 我们还统计了这两个数据集

中跳转表数量与指令数量的比例, 以此表示数据集中跳转表的密度, 密度越大则代表单位指令内包含的跳转表个数越高. 平均来看, SCsmith 生成的数据集中每 1 902 条指令就包含 1 个跳转表, 而文献 [49,50] 包含的数据集中每 4 500 条指令才包含 1 个跳转表. 我们将上述对比结果总结到表 8.

表 8 SCsmith 生成数据集与文献 [49,50] 中真实程序数据集对比

对比维度	SCsmith	文献[49,50]
跳转表个数多	√	×
跳转表生成效率高	√	×
跳转表密度大	√	×
构建是否自动化	√	×

另外, 我们比较了数据集的跳转表大小分布, 结果如图 11 所示. 其中, 由 SCsmith 生成测试集的跳转表平均大小为 25, 中位数是 20, 最小是 4, 最大是 250; 真实程序构成的数据集的跳转表平均大小是 47, 中位数是 16, 最小是 2, 最大是 1 033. 我们发现, SCsmith 生成的数据集的最小值和中位数都与真实程序构成的数据集接近. 然而, 最大值和平均大小与真实程序差距较大. 这是因为为了生成效率, 在 SCsmith 生成 C 程序时, 我们将每个函数的大小设置为最大 128 个语句, 这也就限制了跳转表的大小最大不会超过 256. 为此, 我们将函数的大小设置为最大 1 024 个语句进行 2 h 生成测试, 发现最终生成了 7 个大小超过 1 024 的跳转表, 表明 SCsmith 可以生成与真实程序大小相近的跳转表.

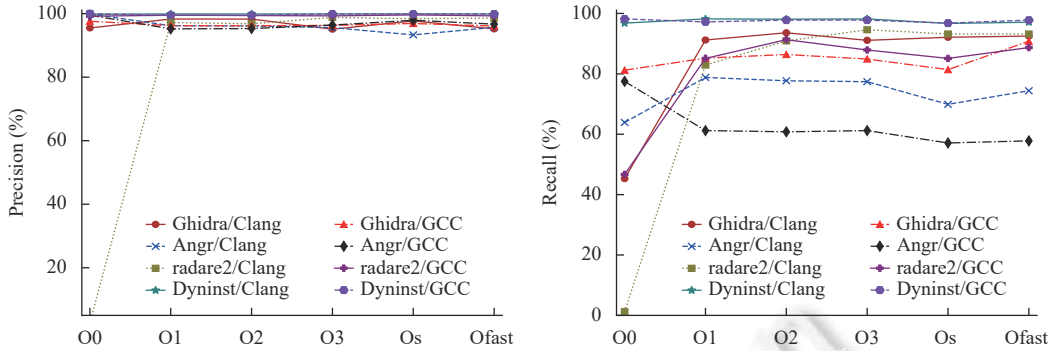


(a) SCsmith 生成数据集 (b) 真实程序构成的数据集

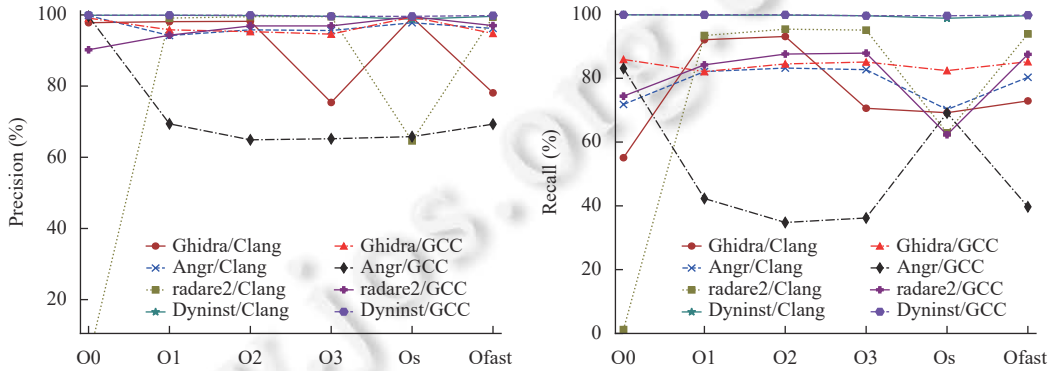
图 11 跳转表大小分布图

6.2 反汇编工具性能评估

使用数据集测试反汇编工具的性能一般有两个目的: 1) 对主流反汇编工具进行对比, 以方便用户选出性能最好的反汇编工具; 2) 发现反汇编工具存在的错误, 进而帮助反汇编工具开发者改进工具的性能. 我们已在第 5.3 节和第 5.4 节展示了这两点. 除此之外, 我们的工作可以作为真实程序数据集的补充, 发现更多反汇编工具算法和实现的错误. 同时, 由于数据集构建和测试的自动化, 补充测试可以持续性进行. 我们在图 12 中展示了使用前期工作[49,50]中真实程序数据集评估反汇编工具性能的结果. 根据图 8 和图 12, 我们计算了反汇编工具在这两个数据集上的整体性能 ($F1\text{-score} = \dfrac{2\times \text{Precision}\times \text{Recall}}{\text{Precision}+\text{Recall}}$ [60]), 计算的结果如表 9 所示. 对比 SCsmith 与前期工作[49,50]可以发现, 不同的反汇编在不同数据集上表现有差异: 比如 Ghidra 和 Dyninst 在 SCsmith 上展示的整体性能与在真实程序构建的数据集上相比有所下降, 这说明了这两个工具在 SCsmith 数据集中存在更多的问题. 将类似的差异与真实数据集结合, 可以发现更多的反汇编工具的问题. 而由于 SCsmith 数据集的生成和测试是自动化的, 可以持续性发现反汇编工具存在的问题.



(a) x86 架构



(b) x64 架构

图 12 反汇编工具在文献 [49,50] 测试集上的性能比较

表 9 反汇编工具在不同测试集上的整体性能

数据集	Angr	Ghidra	radare2	Dyninst
SCsmith	77.9	81.4	85.9	97.9
文献[49,50]	76.3	88.2	77.8	99.1

7 总 结

为了研究开源反汇编工具中求解间接跳转表的现状, 本文从以下两个维度进行了研究: (1) 从启发式算法与可靠性算法两个方面对主流开源反汇编工具使用的算法进行系统性总结; (2) 对这些反汇编工具求解间接跳转表的性能进行自动化评估, 并结合算法实现原理来人工分析反汇编工具求解跳转表出错的原因, 在该过程中我们发现了 6 个反汇编工具在实现上的 bugs. 为了自动化评估跳转表的性能, 我们构建了自动化测试框架: 该框架首先自动化生成包含 switch-case 的 C 文件, 接着经由 OracleGT 编译器自动化生成跳转表 Ground Truth, 最后将 Ground Truth 与反汇编工具的结果进行对比, 从而生成性能报告. 该框架既能够评估反汇编工具的整体性能, 帮助研究人员更好地选择合适的工具; 又可以持续地发现反汇编工具的错误, 帮助反汇编工具开发人员改进工具性能.

References:

[1] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity principles, implementations, and applications. ACM Trans. on Information and System Security, 2009, 13(1): 4. [doi: 10.1145/1609956.1609960]

[2] Zhang M, Sekar R. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In: Proc. of the 31st Annual Computer Security Applications Conf. Los Angeles: ACM, 2015. 91–100. [doi: 10.1145/2818000.2818016]

[3] Zhang C, Wei T, Chen ZF, Duan L, Szekeres L, McCamant S, Song D, Zou W. Practical control flow integrity and randomization for binary executables. In: Proc. of the 2013 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2013. 559–573. [doi: 10.1109/SP.2013.44]

[4] Payer M, Barresi A, Gross TR. Fine-grained control-flow integrity through binary hardening. In: Proc. of the 12th Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Milan: Springer, 2015. 144–164. [doi: 10.1007/978-3-319-20550-2_8]

[5] Wang MH, Yin H, Bhaskar AV, Su PR, Feng DG. Binary code continent: Finer-grained control flow integrity for stripped binaries. In: Proc. of the 31st Annual Computer Security Applications Conf. Los Angeles: ACM, 2015. 331–340. [doi: 10.1145/2818000.2818017]

[6] Pappas V, Polychronakis M, Keromytis AD. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proc. of the 2012 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2012. 601–615. [doi: 10.1109/SP.2012.41]

[7] Zhang Z, Xue JF, Zhang JC, Chen T, Tan YA, Li YZ, Zhang QX. Survey on control-flow integrity techniques. Ruan Jian Xue Bao/Journal of Software, 2023, 34(1): 489–508 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6436.htm [doi: 10.13328/j.cnki.jos.006436]

[8] Chandramohan M, Xue YX, Xu ZZ, Liu Y, Cho CY, Tan HBK. BinGo: Cross-architecture cross-OS binary search. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Seattle: ACM, 2016. 678–689. [doi: 10.1145/2950290.2950350]

[9] Feng Q, Zhou RD, Xu CC, Cheng Y, Testa B, Yin H. Scalable graph-based bug search for firmware images. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. Vienna: ACM, 2016. 480–491. [doi: 10.1145/2976749.2978370]

[10] Hu YK, Zhang YY, Li JR, Gu DW. Binary code clone detection across architectures and compiling configurations. In: Proc. of the 25th Int'l Conf. on Program Comprehension. Buenos Aires: IEEE, 2017. 88–98. [doi: 10.1109/ICPC.2017.22]

[11] Bernat AR, Miller BP. Anywhere, any-time binary instrumentation. In: Proc. of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools. Szeged: ACM, 2011. 9–16. [doi: 10.1145/2024569.2024572]

[12] Shoshitaishvili Y, Wang RY, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng SJ, Hauser C, Kruegel C, Vigna G. SOK: (state of) the art of war: Offensive techniques in binary analysis. In: Proc. of the 2016 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2016. 138–157. [doi: 10.1109/SP.2016.17]

[13] Cova M, Felmetsger V, Banks G, Vigna G. Static detection of vulnerabilities in x86 executables. In: Proc. of the 22nd Annual Computer Security Applications Conf. Miami: ACM, 2006. 269–278. [doi: 10.1109/ACSAC.2006.50]

[14] Li Z, Zou DQ, Xu SH, Ou XY, Jin H, Wang SJ, Deng ZJ, Zhong YY. VulDeePecker: A deep learning-based system for vulnerability detection. In: Proc. of the 25th Annual Network and Distributed System Security Symp. San Diego: NDSS, 2018. [doi: 10.14722/ndss.2018.23158]

[15] Pewny J, Garmany B, Gawlik R, Rossow C, Holz T. Cross-architecture bug search in binary executables. In: Proc. of the 2015 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2015. 709–724. [doi: 10.1109/SP.2015.49]

[16] Meng XZ, Miller BP. Binary code is not easy. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. Saarbrücken: ACM, 2016. 24–35. [doi: 10.1145/2931037.2931047]

[17] University of California. Angr. 2016. https://Angr.io/

[18] NSA. Ghidra. 2019. https://Ghidra-sre.org/

[19] radareorg. radare2. 2021. https://github.com/radareorg/radare2

[20] University of Maryland. Dyninst. 2016. https://www.Dyninst.org/

[21] Yang XJ, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Jose: ACM, 2011. 283–294. [doi: 10.1145/1993498.1993532]

[22] GNU. GCC, The GNU compiler collection. 2023. https://gcc.gnu.org/

[23] LLVM Community. 2023. The LLVM compiler infrastructure. https://llvm.org/

[24] Wikipedia. Ground Truth. 2023. https://en.wikipedia.org/wiki/Ground_truth

[25] Stephens N, Grosen J, Salls C, Dutcher A, Wang RY, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. San Diego: NDSS, 2016. [doi: 10.14722/ndss.2016.23368]

[26] Peng H, Shoshitaishvili Y, Payer M. T-Fuzz: Fuzzing by program transformation. In: Proc. of the 2018 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2018. 697–710. [doi: 10.1109/SP.2018.00056]

[27] Chen YH, Mu DL, Xu J, Sun ZC, Shen WB, Xing XY, Lu L, Mao B. PTrix: Efficient hardware-assisted fuzzing for COTS binary. In: Proc. of the 2019 ACM Asia Conf. on Computer and Communications Security. Auckland: ACM, 2019. 633–645. [doi: 10.1145/3321705.

3329828]

[28] Zhao L, Duan Y, Yin H, Xuan JF. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In: Proc. of the 26th Annual Network and Distributed System Security Symp. San Diego: NDSS, 2019. [doi: 10.14722/ndss.2019.23504]

[29] Muench M, Nisi D, Francillon A, Balzarotti D. Avatar^2: A multi-target orchestration platform. In: Proc. of the 2018 Workshop on Binary Analysis Research. San Diego: BAR, 2018. [doi: 10.14722/bar.2018.23017]

[30] Al3xtjames. Ghidra firmware utilities. 2022. https://github.com/al3xtjames/Ghidra-firmware-utils

[31] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-aware evolutionary fuzzing. In: Proc. of the 24th Annual Network and Distributed System Security Symp. San Diego: NDSS, 2017. [doi: 10.14722/ndss.2017.23404]

[32] Kilgallon S, De La Rosa L, Cavazos J. Improving the effectiveness and efficiency of dynamic malware analysis with machine learning. In: Proc. of the 2017 Resilience Week. Wilmington: IEEE, 2017. 30–36. [doi: 10.1109/RWEEK.2017.8088644]

[33] De La Rosa L, Kilgallon S, Vanderbruggen T, Cavazos J. Efficient characterization and classification of malware using deep learning. In: Proc. of the 2018 Resilience Week. Denver: IEEE, 2018. 77–83. [doi: 10.1109/RWEEK.2018.8473556]

[34] Hernandez G, Fowze F, Tian D, Yavuz T, Butler KRB. FirmUSB: Vetting USB device firmware using domain informed symbolic execution. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. Dallas: ACM, 2017. 2245–2262. [doi: 10.1145/3133956.3134050]

[35] Alasmary H, Anwar A, Park J, Choi J, Nyang D, Mohaisen A. Graph-based comparison of IoT and Android malware. In: Proc. of the 7th Int'l Conf. on Computational Data and Social Networks. Shanghai: Springer, 2018. 259–272. [doi: 10.1007/978-3-030-04648-4_22]

[36] Chen P, Xu J, Hu ZS, Xing XY, Zhu MH, Mao B, Liu P. What you see is not what you get! Thwarting just-in-time ROP with chameleon. In: Proc. of the 47th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Denver: IEEE, 2017. 451–462. [doi: 10.1109/DSN.2017.47]

[37] Miller BP, Christodorescu M, Iverson R, Kosar T, Mirgorodskii A, Popovici F. Playing inside the black box: Using dynamic instrumentation to create security holes. Parallel Processing Letters, 2001, 11(2–3): 267–280. [doi: 10.1142/S0129626401000579]

[38] Armstrong W, Christen P, McCreath E, Rendell AP. Dynamic algorithm selection using reinforcement learning. In: Proc. of the 2006 Int'l Workshop on Integrating AI and Data Mining. Hobart: IEEE, 2006. 18–25. [doi: 10.1109/AIDM.2006.4]

[39] Mußler J, Lorenz D, Wolf F. Reducing the overhead of direct application instrumentation using prior static analysis. In: Proc. of the 17th Int'l European Conf. on Parallel Processing. Bordeaux: Springer, 2011. 65–76. [doi: 10.1007/978-3-642-23400-2_7]

[40] Sidiroglou S, Laadan O, Perez C, Viennot N, Nieh J, Keromytis AD. ASSURE: Automatic software self-healing using rescue points. ACM SIGARCH Computer Architecture News, 2009, 37(1): 37–48. [doi: 10.1145/2528521.1508250]

[41] Cifuentes C, Van Emmerik M. Recovery of jump table case statements from binary code. Science of Computer Programming, 2001, 40(2–3): 171–188. [doi: 10.1016/S0167-6423(01)00014-4]

[42] LLVM Community. MachineJumpTableInfo class reference. 2023. https://llvm.org/docs/doxygen/classllvm_1_1MachineJumpTableInfo.html

[43] Balakrishnan G, Reps T. Analyzing memory accesses in x86 executables. In: Proc. of the 13th Int'l Conf. on Compiler Construction. Barcelona: Springer, 2004. 5–23. [doi: 10.1007/978-3-540-24723-4_2]

[44] Balakrishnan G, Reps T. WYSINWYX: What you see is not what you eXecute. ACM Trans. on Programming Languages and Systems, 2010, 32(6): 23. [doi: 10.1145/1749608.1749612]

[45] Kinder J, Veith H. Jakstab: A static analysis platform for binaries. In: Proc. of the 20th Int'l Conf. on Computer Aided Verification. Princeton: Springer, 2008. 423–427. [doi: 10.1007/978-3-540-70545-1_40]

[46] Williams-King D, Kobayashi H, Williams-King K, Patterson G, Spano F, Wu YJ, Yang JF, Kemerlis VP. Egalito: Layout-agnostic binary recompilation. In: Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2020. 133–147. [doi: 10.1145/3373376.3378470]

[47] LLVM Community. LLVM's analysis and transform passes. 2022. https://llvm.org/docs/Passes.html

[48] Andriesse D, Chen X, Van Der Veen V, Slowinska A, Bos H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: Proc. of the 25th USENIX Conf. on Security Symp. Austin: USENIX Association, 2016. 583–600. [doi: 10.5555/3241094.3241140]

[49] Pang CB, Yu RT, Chen YH, Koskinen E, Portokalidis G, Mao B, Xu J. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In: Proc. of the 2021 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2021. 833–851. [doi: 10.1109/SP40001.2021.00012]

[50] Pang CB, Zhang TT, Yu RT, Mao B, Xu J. Ground truth for binary disassembly is not easy. In: Proc. of the 31st USENIX Security Symp. Boston: USENIX, 2022. 2479–2495.

[51] Jiang MH, Dai QM, Zhang WL, Chang R, Zhou YJ, Luo XP, Wang RY, Liu Y, Ren K. A comprehensive study on ARM disassembly

tools. IEEE Trans. on Software Engineering, 2023, 49(4): 1683–1703. [doi: 10.1109/TSE.2022.3187811]
[52] Bigot PA, Debray S. Return value placement and tail call optimization in high level languages. The Journal of Logic Programming, 1999, 38(1): 1–29. [doi: 10.1016/S0743-1066(98)80001-0]
[53] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984, SE-10(4): 352–357. [doi: 10.1109/TSE.1984.5010248]
[54] LLVM Community. The “MC” layer. 2023. https://llvm.org/docs/CodeGenerator.html#the-mc-layer
[55] GNU. GNU binutils. 2023. https://www.gnu.org/software/binutils/
[56] GNU. GCC. RTL representation. 2022. https://gcc.gnu.org/onlinedocs/gccint/RTL.html
[57] Bartlett J. Common and useful assembler directives. In: Bartlett J, ed. Learn to Program with Assembly: Foundational Learning for New Programmers. Berkeley: Apress, 2021. 165–171. [doi: 10.1007/978-1-4842-7437-8]
[58] NSA. P-Code reference manual. 2017. https://spinsel.dev/assets/2020-06-17-Ghidra-brainfuck-processor-1/Ghidra_docs/language_spec/html/pcoderef.html
[59] NSA. P-Code operation reference. 2020. https://spinsel.dev/assets/2020-06-17-Ghidra-brainfuck-processor-1/Ghidra_docs/language_spec/html/pcodedescription.html#cpui_int_negate
[60] Sasaki Y. The truth of the *F*-measure. 2007. https://www.cs.odu.edu/~mukka/cs795sum09dm/Lecturenotes/Day3/F-measure-YS-26Oct07.pdf

附中文参考文献:

[7] 张正, 薛静锋, 张静慈, 陈田, 谭毓安, 李元章, 张全新. 进程控制流完整性保护技术综述. 软件学报, 2023, 34(1): 489−508. http://www.jos.org.cn/1000-9825/6436.htm [doi: 10.13328/j.cnki.jos.006436]

庞成宾(1995－), 男, 博士, 主要研究领域为逆向工程, 漏洞挖掘, 软件保护.

张天泰(1998－), 男, 硕士生, 主要研究领域为逆向工程.

徐雪兰(1998－), 女, 硕士, 主要研究领域为逆向工程, 漏洞挖掘.

茅兵(1967－), 男, 博士, 教授, 博士生导师, CCF专业会员, 主要研究领域为逆向工程, 漏洞挖掘, 软件安全防护.