

西安交通大学

硕士学位论文

反编译中的程序结构优化研究

学位申请人：刘延昭

指导教师：赵银亮 教授

学科名称：计算机科学与技术

2015 年 05 月

Research of Program Structure Optimization in Decompilation

A thesis submitted to
Xi'an Jiaotong University
in partial fulfillment of the requirements
for the degree of
Master of Engineering

By
Yanzhao Liu
Supervisor: Prof. Yinliang Zhao
Computer Science and Technology
May 2015

论文题目：反编译中的程序结构优化研究

学科名称：计算机系统结构

学位申请人：刘延昭

指导教师：赵银亮 教授

摘 要

随着智能设备的广泛使用以及互联网和物联网的普及，代码的安全问题越来越突出，理解并监督程序势在必行。反编译做为程序理解的重要辅助手段之一，其输出程序的控制结构优劣直接影响着程序的可读性，进而影响程序理解。

本文^①分析编译优化对程序结构的改变，总结主流反编译器的程序结构化算法，并结合图论中的相关算法来研究反编译中程序结构优化问题。分支结构的优化主要集中在基于跳转表的 switch 结构和基于二叉树的 switch 结构的重构和规整。程序中的 goto 语句往往会增加程序理解的复杂度，基于代码复制消除 goto 语句的方法，可以在一定程度上消除 goto 语句并规范代码结构。编译器为了提高生成代码的执行效率，通常会将固有函数调用转换成内联函数，在反编译中将内联固有函数适当地进行规约和消除可以为后续的反编译流程提供重要的类型信息，并且能够简化后续数据流分析和控制流分析，从而提高反编译结果的可读性。通过对比各种反编译器设计方案，提出并实现 ASMBoom 反编译器，其最大特点是在中间代码生成之前，在基于控制流图的汇编程序上检测内联固有函数和指令习语，从而提高中间代码的抽象程度。

以 SPEC CPU 2006 标准测试集，Olden 标准测试集和 Github 上的部分开源程序作为测试用例，以 Hex-Rays，RD，Boomerang 和 REC 等主流反编译器作为对比平台，评价 ASMBoom 反编译器的整体性能和部分结构优化算法的效果。实验结果表明，ASMBoom 在 PowerPC（ppc）程序的反编译方面要优于 Boomerang 和 RD 反编译器。在结构优化算法方面，ASMBoom 在 ppc 的 switch 结构的重构和规整要优于其它反编译器。在 goto 语句消除方面，ASMBoom 比 Boomerang 消除了更多的 goto 语句。在内联固有函数消除方面，ASMBoom 能够消除比其它反编译器更多的内联固有函数。

关 键 词：反编译；编译优化；程序结构优化；图论；ASMBoom 反编译器

论文类型：应用基础

^① 本研究得到企业委托项目(201312127)和陕西省科技计划项目(2014K05-04)的资助。

Title: Research of Program Structure Optimization in Decompilation

Discipline: Computer Architecture

Applicant: Yanzhao Liu

Supervisor: Prof. Yinliang Zhao

ABSTRACT

With the extensive usage of smart devices and the prevalence of internet and the Internet of Things, the security of code is becoming more and more outstanding so that understanding and supervision of programs is inevitable. Decompilation is one of the most important facility tools in program understanding. The merits of the program structure affect the readability and understanding of the decompilation output.

This paper analyzes the impact of compiler optimization on the program structure, summarizes the control flow structure algorithms on mainstream decompilers and combines algorithms in graph theory to solve the problems of program structure optimization in decompilation. The program structure optimization in decompilation is focused on the optimization of multi-branches, the elimination of goto statement and the inlined intrinsic functions. Multi-branches optimizations mainly considered the reconstruction and the regulation of switch structure based on jump table and switch structure based on binary tree. Goto statement can increase the complexity of the program understanding, goto elimination based on code duplication can regulate the program structure to some extent. Compilers always inline some intrinsic functions to improve the speed of the code. Eliminating inlined intrinsic functions earlier can provide more type information, simplify the data flow analysis and control flow analysis during the decompilation process and improve the readability of the decompilation output. We design the ASMBoom decompiler based on the designs of other decompilers. The biggest advantage is that intrinsic function elimination and instruction idiom is analyzed on CFG based assembly program before the IR is generated.

We use the SPEC CPU 2006 testsuite, Olden testsuite and some programs on Github as the testsutes. Using Hex-Rays, RD, Boomerang and REC as the comparison platform to evaluate the overall performance of ASMBoom and the effects of the program structure optimization algorithms. The experiment results show that ASMBoom is much superior to Boomerang and RD on ppc decompilation. ASMBoom outperforms other decompilers on the reconstruction and regulation of switch structure. With respect to the elimination of goto statements, ASMBoom eliminates more gotos than Boomerang. On the elimination of inlined intrinsic function, ASMBoom reduces more inlined functions than other decompilers.

KEY WORDS: Decompilation; Compiler Optimization; Program Structure Optimization; Graph Theory; ASMBoom Decompiler

ABSTRACT

TYPE OF THESIS: Application Fundamentals

目 录

1 绪论	1
1.1 研究背景	1
1.1.1 逆向工程与反编译	1
1.1.2 反编译技术的应用	2
1.1.3 编译优化与反编译优化	3
1.2 本文的主要工作	4
1.3 论文的组织结构	5
2 反编译中程序结构优化的相关研究	6
2.1 反编译器及其关键技术的研究	6
2.1.1 Boomerang 反编译器	6
2.1.2 Retargetable Decompiler 反编译器	6
2.1.3 Phoenix 反编译器	7
2.1.4 基于搜索的反编译器	8
2.1.5 Hex-Rays 反编译器	9
2.2 编译对程序结构的优化	9
2.2.1 分支结构的优化	9
2.2.2 循环结构的优化	11
2.2.3 函数内联	12
2.3 反编译中的结构优化	13
2.3.1 Goto 语句优化	13
2.3.2 控制流图的结构化	14
2.4 图论的相关研究	17
2.4.1 图同构	17
2.4.2 图拓扑嵌入	18
2.5 本章小结	19
3 程序结构优化	20
3.1 反编译器框架	20
3.1.1 智能解码器	21
3.1.2 中间表示生成	24
3.2 Switch 结构的优化技术	25
3.2.1 基于跳转表的 switch 结构的优化	25
3.2.2 基于二叉树的 switch 结构的优化	27
3.3 Goto 语句消除技术	30

3.3.1 Goto 语句分析	31
3.3.2 代码复制消除 goto 语句	31
3.4 固有函数的消除技术	34
3.4.1 固有函数分析	34
3.4.2 基于子图同构的内联固有函数消除算法	35
3.4.3 基于图拓扑嵌入的内联固有函数消除	40
3.5 本章小结	41
4 ASMBoom 反编译器实现	42
4.1 反编译器实现	42
4.1.1 Boomerang 反编译器	43
4.1.2 ASMBoom 前端工作流程	46
4.1.3 可扩展的前端	47
4.2 固有函数消除算法实现	48
4.2.1 固有函数模板库构造	48
4.2.2 Vflib 库介绍	49
4.3 指令习语数据库构建	52
4.4 Switch 结构优化实现	53
4.5 Goto 语句消除实现	55
4.6 本章小结	57
5 实验测试及分析	58
5.1 实验方法	58
5.2 反编译器整体性能测试	58
5.3 程序结构优化测试	60
5.4 反编译结果实例	62
5.5 本章小结	66
6 结论与展望	67
6.1 结论	67
6.2 进一步工作展望	68
致 谢	69
参考文献	70
攻读学位期间取得的研究成果	75
声明	

CONTENTS

1	Introduction	1
1.1	Research Background.....	1
1.1.1	Reverse Engineering and Decompilation	1
1.1.2	Application of Decompilation	2
1.1.3	Compile Optimization and Decompile Optimization.....	3
1.2	Thesis Contents	4
1.3	Thesis Organization.....	5
2	Releted Work of Program Structure Optimization in Decompilation	6
2.1	Decompiler and its Key Techniques.....	6
2.1.1	Boomerang Decompiler	6
2.1.2	Retargetable Decompiler.....	6
2.1.3	Phoenix Decompiler.....	7
2.1.4	Decompiler based on Search	8
2.1.5	Hex-Rays Decompiler	9
2.2	Program Structure Optimization of Compile	9
2.2.1	Optimization of Branch Structure	10
2.2.2	Optimization of Loop	11
2.2.3	Function Inline	12
2.3	Program Structure Optimization of Decompile.....	13
2.3.1	Optimization of Goto Statement.....	13
2.3.2	Structure Control Flow Graph.....	14
2.4	Related Work of Graph Theory	17
2.4.1	Graph Isomorphisim.....	17
2.4.2	Topological Embedding of Graph	18
2.5	Brief Summary	19
3	Program Structure Optimization	20
3.1	Framwork of Decompiler	20
3.1.1	Intelligent Decoder	21
3.1.2	IR Generation	24
3.2	Optimizations of Switch Structure	25
3.2.1	Optimization of Jump Table based Switch Structure	25
3.2.2	Optimization of Binary Tree based Switch Structure.....	27
3.3	Goto Statement Elimination Techniques	30
3.3.1	Analysis of Goto Statement.....	31
3.3.2	Goto Elimination based on Code Duplication.....	31
3.4	Elimination of Inlined Intrinsic Function	34
3.4.1	Analysis of Intrinsic Function	34
3.4.2	Inline Intrinsic Function Elimination based on Graph Isomorphisim	35

3.4.3 Inline Intrinsic Function Elimination based on Topological Embedding.....	39
3.5 Brief Summary	40
4 Implementation of ASMBoom Decompiler	42
4.1 Implementation of Decompiler	42
4.1.1 Boomerang Decompiler	43
4.1.2 Workflow of the Frontend of the ASMBoom Decompiler	45
4.1.3 Extensible Frontend.....	47
4.2 Implementation of Intrinsic Function Elimination	48
4.2.1 Construction of the Intrinsic Function Templates	48
4.2.2 Vflib Library Introduction.....	49
4.3 Construction of Instruction Idiom Database	51
4.4 Implementation of switch structure optimization.....	53
4.5 Implementation of goto statement elimination.....	55
4.6 Brief Summary	57
5 Experimental Results and Analysis	58
5.1 Experiment Approach.....	58
5.2 Testing of the Overall Performance of ASMBoom	58
5.3 Testing of Program Structure Optimization	60
5.4 Decompilation Results Illustration	62
5.5 Brief Summary	66
6 Conclusions and Suggestions	67
6.1 Conclusions	67
6.2 Future Work.....	68
Acknowledgements	69
References	70
Achievements	75
Declaration	

1 绪论

1.1 研究背景

智能设备的广泛使用，伴随着互联网和物联网的迅速发展，使得世界越来越扁平化；程序越来越复杂，程序的总量迅猛增加，程序间的交互广泛，人类的活动越来越依赖程序，程序可靠则人类生活有保障，程序失信则人类生活遭殃。但是，由于程序本身和对程序的监管不足等原因，人类由此遭受的损失也越来越大。2014 年，据阿里巴巴聚安全报道，对 11 个类别的热门 app 进行静态扫描，参与测试的 app 中近 97% 的 app 都有安全漏洞，且平均漏洞量达 40 个。另外，据美国国家标准与技术研究院(NIST) 2002 年的研究报告称，程序缺陷每年造成近 599 亿美元的损失，接近美国 0.6% 的 GDP。保护用户利益，程序监管势在必行。

作为程序监管的主要技术手段，程序理解和代码验证成为必然选择。程序理解从程序的静态或动态运行的角度分析其特征和功能，作为代码验证的主要依据。事先排除恶意代码，规范程序行为，对保护用户利益和净化软件市场都有着重要的意义。

反编译作为程序理解和代码验证的一种有效手段，旨在从二进制代码或汇编代码中恢复出高级程序，辅助逆向分析人员理解程序的功能。提高高级程序的可读性，成为反编译的重要研究方向，如程序控制结构的恢复和优化，程序中基本类型和聚合类型的恢复，程序中基本操作的组合和优化，程序中指令习语的消除等。本文重在研究反编译过程中程序控制结构的恢复和优化。

1.1.1 逆向工程与反编译

逆向工程是一种分析软件的组件以及各个组件的交互方式，并将软件用更抽象的方式描述以辅助人理解的过程。由此我们能够更好地了解软件的结构和其具体的操作。逆向工程是一个实践性非常强的工程领域，在与安全相关的逆向领域，可以用来分析恶意软件，逆向加密算法和审计二进制代码等；在正常的软件开发方面，逆向工程可以用来分析正版软件和评估软件的质量和鲁棒性。

目前，常用的逆向分析工具有监视器、反汇编器、调试器和反编译器。监视器用来收集被分析程序与其环境的交互信息并显示。FileMon、TCPView、RegMon、PortMon等都是常见的分别用于监视程序与系统文件的交互情况、TCP 和 UDP 网络连接情况、注册表的访问情况和物理端口状态的工具。反汇编器通过线性扫描或递归扫描技术^[1]分析二进制中的指令流序列，并将其翻译成可读性更高的汇编代码。IDA Pro 和 ILDasm 是工业界使用地比较多的反汇编器；在学术界，Capstone 为多平台多体系结构的反汇编提供了一个框架，被广泛的使用^[2]。调试器在逆向分析领域用来观察程序的动态行为，通过设置程序断点，获取当前寄存器和内存的状态，更加深入地理解程序的行为。其不足之处在于，由于输入的限制，在特定输入情况下调试程序，只有部分代码被执行，

因此分析不是很全面。常见的用户级的调试器有 OllyDbg、WinDbg 和 IDA Pro 等。

反编译器作为一种静态的，全面的逆向分析工具，旨在将二进制代码或汇编代码转换成可读性更高的高级代码。常见的反编译器利用编译理论，通过控制流图构建，数据流分析，类型分析，控制流分析和高级代码生成等步骤完成机器代码向高级程序的转换。基于编译理论的反编译器始于 DCC^[3]的创始人 Cristina Cifuentes。最近，出现了基于搜索的反编译器，其详细内容将在 2.1.4 节展开介绍。

目前，反编译器的发展方向概括如下：

首先，反编译器所采用的中间表达式在向 LLVM IR 等通用中间表达式靠拢，如 Binary Analysis Platform (BAP)^[4]、SecondWrite^[5]、Retargetable Decompiler (RD)^[6]、Fracture^[7]和 Libbeauty 等。可见 LLVM 编译器^[8]的程序终身代码优化特性和 LLVM IR 的 SSA 特性以及独立于具体语言的类型系统对于反编译的研究和开发有巨大的影响。

其次，反编译的应用领域在扩展，如 SecondWrite 将反编译应用到二次代码优化中，从而提高遗产系统在新的并行执行体系结构上的执行效率。

再者，反编译不再局限于二进制代码。随着 .NET 和 JAVA 的广泛使用，MSIL 和 JAVA 字节码的反编译对于审计和定制相关应用程序也非常重要。

最后，反编译器开发的门槛逐步降低，但是反编译结果的优化越来越困难。开源社区对于反编译器的开发影响深远。基于编译理论的反编译结果优化逐渐显得捉襟见肘，需要辅之以数据挖掘等方面的知识。

为提高反编译结果的可读性，首先需了解编译器对高级代码的优化，再根据具体优化特点，逐步去除优化效果，从而更加有针对性地提出反编译的优化算法，既包括结构方面的优化也包括指令习语方面的优化。3.1.1 节将从结构和指令习语的优化方面入手，介绍智能解码器。

1.1.2 反编译技术的应用

反编译自从 20 世纪 60 年代诞生以来，主要被用于代码迁移，代码理解，代码维护和代码安全验证等方面。

在代码迁移方面，可以实现代码在不同平台之间的迁移，从而兼容和有效利用目标平台的各种新特性。SecondWrite 可以将二进制转换成 LLVM IR，对中间语言进行自动并行化分析和优化，再次生成二进制代码。实验结果表明，SecondWrite 对未经过优化的二进制代码进行二次优化后的加速比可以提高 27%。Emscripten^[9]是一款将 LLVM IR 中间表达式翻译成 Javascript 的编译器。Emscripten 的重要意义在于为桌面应用程序向 Web 运行环境迁移提供了重要的技术支持。将 Fracture 反编译器和 Emscripten 进行组合，就可以搭建实现桌面应用程序向 Web 运行环境迁移的反编译器雏形。

在代码理解方面，反编译产生的高级代码更容易让人理解程序的逻辑和算法。根据[10]对市场上流行的 1100 款 Android 应用程序进行反编译分析，发现存在大量的个人 ID 和手机 ID 的误用，广告的深层渗透等不合法行为。代码中所蕴含的不为人知的存在安全隐患的库函数和算法的使用，缓冲区溢出，整数溢出和缺乏对输入数据的验

证等缺陷都会给用户造成重大的损失。2013 年，由于通用的 `plug-and-play`^[11] 库中的一个缓冲区溢出错误，造成了互联网中有将近 2300 万路由器处在安全隐患之中。可见在互联网环境下，程序在逻辑和算法上的一个微小的错误，就能引发互联网世界的惊涛巨浪。

在代码维护方面，当源码不可用时，仍然可以通过反编译手段，重新构建软件^[12]，实现软件定制和二次开发。

在代码安全验证方面，反编译技术可以用来验证编译器产生的目标代码，检查软件中是否存在恶意代码等。编译器作为一款普通的软件，如同其它软件一样，其中必然存在 bug。例如在优化过程中会产生错误的目标代码，NULLSTONE 对 20 款商业编译器在整数除法的优化进行评估，发现有 12 款编译器存在瑕疵。同时，Xuejun Yang 等^[13]开发了 Csmith 系统，用于随机生成测试用例测试现有 c 编译器，累积报告了 325 个之前不知道的编译器 bug；同时，编译器的源码在编译的时候会被插入恶意代码，从而为编译器引入漏洞。最著名的例子^[14]是 Thompson 为 UNIX 系统写的 login 程序，它允许任何一个有后门密码的人登录并控制系统。软件审计人员不能通过检查 login 程序的源码分析出原因。该漏洞是由经过特殊处理的 C 编译器引入。

目前，反编译的应用领域主要集中在与安全相关的工业领域和国防领域。美国空军研究实验室赞助的 Fracture 开源反编译器项目，实现 x86, ppc 和 arm 指令系统的反编译。随着嵌入式设备的广泛应用，国民经济的各个领域越来越依赖于程序。在源码不可得或源码丢失的情况下，反编译可以很好的分析可执行程序算法及其安全性等。

1.1.3 编译优化与反编译优化

反编译的输入一般是经过编译器优化之后的机器代码，因此提高反编译结果的可读性，必须先了解编译优化算法。编译的优化约束条件是特定的目标机的有限的计算资源，其优化目标是目标代码的运行时间最优或目标代码的体积最优或二者兼顾等。在这样一个优化问题里，不同的编译器根据不同的目标机，对程序的控制结构和程序的数据流进行优化，以便达到优化的目的。

基本块的重新排序可以消除代码中的条件跳转数目，从而提高代码的局部性。不可到达代码删除从程序入口点出发，深度遍历整个控制流图，删除基本块列表中没有被遍历到的基本块，从而达到缩减代码体积的目的。循环融合将多个循环融合成一个循环。循环展开通过增大循环体，减少循环迭代次数，有效提高代码和数据的局部性，达到代码优化的目的。循环外提将循环中的 if 分支提到循环外边并与原有循环组成新的循环，将剩余部分再组织成新的循环，构成 else 分支，从而实现循环的分离，提高并行能力。循环分裂将一个循环分裂成多个循环，该种优化算法可以使程序在多核体系结构上运行加速。

Switch 结构的优化根据 case 分支的值分布，分别可以被翻译成基于二叉树的嵌套 if-else 结构或基于跳转表的结构等。同时，在反编译过程中，switch 结构的解码属于间接跳转指令的解码，在实际操作中需要正确处理，从而有效构造被反编译过程体的控

制流程图，为后续的反编译优化提供方便。尾合并将 `if-else` 结构中的公共代码块提取出来，并放置在其后续基本块中，从而达到精简代码体积的目的。

函数内联根据被调用函数的申明情况，被内联函数的指令数目以及内联后对编译单元大小的影响等因素来确定一个过程体是否被内联。内联的好处在于减少函数调用的开销，如栈的初始化等。宏展开是在编译预处理阶段，将由宏定义的表达式或语句就地替换的过程。从反编译的角度来看，二者都增加了反编译过程中数据流和控制流分析的复杂度。

在数据流的优化方面，指令习语的使用虽然会达到编译优化的目标。但是，指令习语对于反编译来说却非常具有挑战。其根本原因在于，指令习语是面向机器的，有许多非常细节的操作，可读性非常低。

上述的编译优化，有些对提高反编译结果的可读性是有益的，如基本块的重排序，不可到达代码删除，循环融合和尾合并等。有些编译优化对反编译结果的可读性是不利的，如循环展开，循环外提，循环分裂，`switch` 结构的优化，函数内联，宏展开和指令习语等。研究具体的编译优化对于反编译中的优化目标和方向有重要指导意义。

数据流分析，类型分析和控制流分析是反编译中的三大关键组成部分。数据流分析中涉及函数以及被调用过程体的参数以及返回值的分析，无用代码删除，表达式的组合和变量重命名等；类型分析中涉及基本类型的获取以及类型的推导过程；控制流分析涉及控制流图的规整以及控制流图的结构化，其中控制流图的规整需要广泛借鉴编译优化中对各种程序控制结构（顺序，分支和循环）的优化方法，从而逆向恢复出原始的控制结构，如 `switch` 结构的规整以及条件表达式组合，都需要研究相应的编译优化算法，才能设计对应的反编译规整算法。指令习语的分析更需要对编译优化中相应的算法有所了解，才能恢复出具体的抽象操作及其操作数。

总之，反编译过程中的优化算法与编译优化算法是密切相连的。

1.2 本文的主要工作

通过对代码安全局势，以及逆向工程和反编译技术在代码迁移，代码理解，代码维护和代码验证等方面的应用分析，突出反编译技术的重要作用。反编译结果的可读性是反编译研究的重要内容之一，而反编译的输入通常是编译优化后的代码。因此，反编译优化的研究离不开对于编译优化算法的研究。本文结合图论和编译优化等相关知识和技术，着重研究反编译中的程序结构优化。

本文的主要工作如下：

(1) 研究国内外主要反编译器，包括其软件框架，中间表达式和关键优化技术等。同时，对比各个反编译器在软件框架和优化算法等方面的优缺点，提出自己的反编译框架并说明其优点。

(2) 详细分析了编译优化中对程序结构有重大改变的优化算法和指令习语的相关算法，总结具体的结构特征和指令序列流特征，用于设计具体的反编译结构优化算法

和指令习语检测算法。

(3) 深入研究图论中的图同构算法和图拓扑嵌入算法，并分析这些算法在反编译中程序控制结构优化方面的广泛应用。

(4) 设计和实现了反编译结构优化算法，如通过代码复制消除 `goto` 语句，`switch` 结构的规整和优化，基于图同构的内在函数消除和窥孔优化算法实现指令习语的检测等。

(5) 使用 SPEC CPU 2006 标准测试集，Olden 标准测试集和 Github 上的部分开源程序集对 ASMBoom 反编译器进行整体测试和评估。同时，评测论文中提到的各种反编译结构优化算法。

1.3 论文的组织结构

本文共分为六章，具体内容概括如下：

第一章，即本章，对反编译中的程序结构优化研究的背景做了简单介绍。同时，对论文的内容做了简单概括。

第二章，研究和对比主流的反编译器，编译和反编译过程中的程序结构优化算法以及图论中的图同构和图拓扑嵌入的概念及相应算法。

第三章，提出自己的反编译框架以及对 `switch` 结构，`goto` 结构和固有函数的消除技术给出了详细描述。

第四章，介绍反编译器 ASMBoom 的实现技术和指令系统扩展方法。同时，介绍各个结构优化算法的实现。

第五章，实验测试及分析，使用 SPEC CPU 2006 标准测试集，Olden 标准测试集和 Github 上的部分开源代码对 ASMBoom 反编译器进行整体测试与评估。

第六章，结论与展望，提出后续工作目标。

2 反编译中程序结构优化的相关研究

本章先从现有反编译器及其关键技术入手，分析各个反编译器在软件框架和关键技术方面的不足，逐步深入探讨编译和反编译对程序结构的优化。最后，给出可以被广泛使用在反编译优化中的图同构和图拓扑嵌入的概念，算法及其应用。

2.1 反编译器及其关键技术的研究

反编译的研究始于上个世纪 60 年代，发展将近有半个多世纪。但是，反编译真正具有里程碑意义的发展应当归功于昆士兰大学的 Cristina Cifuentes。在她的博士论文^[3]中，她首次提出了基于编译理论的反编译研究方向并给出了在 DOS 操作系统下的针对 i80286 指令系统的二进制程序的反编译器原型 DCC。本章介绍的反编译器，除 2.1.4 节提到的基于搜索的反编译器外，都基于 DCC 反编译器原型。大致包括前端，中间端和后端。前端包括加载器和解码器；中间端包括数据流分析，类型分析和控制流分析；后端包括代码生成。

2.1.1 Boomerang 反编译器

Boomerang^[15]是学术界现有的重要的开源软件，其支持多种体系结构多种文件格式。Boomerang 采用 SLED (Specification Language for Encode and Decode)^[16]语言描述二进制指令中二进制位的语义，用于将二进制序列流转换成汇编指令流；采用 SSL (Semantic Specification Language)^[17]语言描述汇编指令的语义，实现汇编代码到中间代码的转换。中间代码再经过中间端和后端的分析，实现类 C 代码的生成。

在 Boomerang 的影响下，许多开源逆向分析工具采用了 Boomerang 的技术路线，典型的工具包括 IDC (Interactive DeCompilation)^[18]和 Jakstab^[19]。IDC 采用 SSL 描述指令语义，并将汇编指令转换成 RTL (Register Transfer Language) 中间表达式，通过交互式的方法辅助完成死代码删除，控制流图简化，数据流简化，变量重命名和表达式化简等复杂的优化算法，通常可以获得较好的反编译结果。Jakstab 反汇编器采用多遍迭代式扫描和数据流分析相结合的算法，实现 Hex-Rays 所不能完成的间接跳转的翻译。实验结果表明，Jakstab 能够比 Hex-Rays 多恢复出约 25% 的间接跳转。

Boomerang 在控制流图优化方面没有针对 1.1.3 节中的编译优化算法的特殊处理。在结构化算法方面，采用基于区间的结构化算法，详细内容将在 2.3.2 节给出。

根据测试^[6, 20]，Boomerang 反编译软件只支持 $\text{intA} = \text{intB} < 0$ 指令习语的识别。因此，在指令习语方面的工作不足，有待改善。

2.1.2 Retargetable Decompiler 反编译器

RD^[21]是捷克共和国布尔诺科技大学 Lissom 项目的重要组成部分。RD 采用 OFFDL (Object File Format Description Language) 语言描述不同类型的文件，并将二进制应用

程序转换成独立于平台的，统一的中间文件格式 COFF (Common Object File Format)；使用 ISAC 模型对指令系统进行建模，完成具体指令到 LLVM IR 中间语言的转换。这样的前端设计，使得 RD 具有很好的扩展性，可以随意扩展并实现不同体系结构和不同文件类型的反编译器。当程序被转换成中间代码时，RD 会调用 LLVM 编译器的内建优化算法，如循环优化，常量传播和控制流图简化等优化中间代码。最后，通过代码生成模块完成中间程序到高级代码的转换。

目前，大量的逆向分析工具开始采用 LLVM IR 作为中间表达式或者有产生 LLVM IR 的接口。SecondWrite, Fracture, BAP, Dagger^[22], mcsema^[23], RevGen^[98], libbeauty 等反编译器就是非常有代表性的例子。

在指令习语分析方面，RD 采用一种类似于编译器中的窥孔优化的算法。该算法在一个基本块内顺序扫描并检测 LLVM IR 中的指令习语序列，忽略由于指令调度在指令习语中间插入的无关代码，并将这些指令保持不动。目前，该算法不支持跨越基本块的指令习语检测。实验结果表明，在 ARM 程序的指令习语分析中，RD 能够检测出 98.9% 的指令习语，要优于 Hex-Rays 的 58.2% 的检出率；但是，在 x86 程序的指令习语分析中，RD 的检出率只有 21.6%，明显低于 Hex-Rays 的 63.5% 的检出率。

在内联固有函数的分析中，RD^[24]从静态库中抽取通用的特征，并用这些特征去匹配静态代码中被链接到代码中的库函数。这些通用的特征包括库函数的头几个字节，CRC 校验码，模块大小，公共符号和结尾的比特流。同时采用树形结构组织特征库，从而提高匹配速率。

在实践中，RD 被用于病毒分析。最近，在捷克、美国、英国和德国流行的偷窃银行账号和密码的 Vawtrak 木马势头猖獗。AVG 安全小组使用 RD 反编译器，成功分析出了 Vawtrak 木马^[25]的机理并给出了具体防护措施。

2.1.3 Phoenix 反编译器

Phoenix^[26]的反编译流程大致包括控制流图恢复，类型恢复，控制流结构恢复和语句翻译等四大步骤。Phoenix 采用卡内基梅隆大学的 BAP 分析工具完成控制流图中的 x86 汇编指令向 BAP Intermediate Language (BIL) 的转换。在恢复带有间接跳转指令的控制流图时，Phoenix 采用了 VSA (Value Set Analysis)^[27]算法；采用 TIE^[28]完成二进制代码中变量及其类型的恢复。TIE 通过 VSA 算法恢复变量位置，并用静态的基于约束的类型推导系统给每个变量指派类型。

控制流结构恢复是 Phoenix 反编译器中最重要的组成部分。在分析了传统的结构化算法在正确性和结构恢复程度方面的不足之后，Phoenix 提出了一种迭代式精化和保持语义的结构化算法。实验表明，Phoenix 的这种迭代式精化和保持语义的结构化算法能够比传统的结构化算法多恢复出 30 多倍的控制结构，并且正确性是 Hex-Rays 反编译器的两倍之多。

在语句翻译阶段，基本块中的 BIL 被翻译成高级语言 HIL (High Intermediate Language)。同时，采用 VSA 恢复函数的参数；使用死代码删除算法删除冗余代码；

通过手工构造 20 多个组合模板 (untiling patterns) 用来化简 gcc 代码生成器生成的指令序列流。

迭代式精化和保持语义的结构化算法的详细内容将在 2.3.2 节给出。其算法的基本思想是通过迭代的方式，每次迭代过程中，根据基本块的拓扑特征依次匹配非循环模式，switch 模式，循环模式等，直到迭代不再产生变化；在文献中未发现 Phoenix 中有指令习语的分析，这或许是 Phoenix 反编译器的不足之处。

2.1.4 基于搜索的反编译器

鉴于代码重用，Wei Ming Khoo^[29]提出了基于搜索的反编译器。代码重用的最大推动力来自于开源软件运动。据 2011 年一月到五月的代码提交次数显示，Github 有 1,153,059 次提交量，远远超过 Sourceforg, Google Code 和 CodePlex 的提交总和；同时，代码重用商业软件和恶意代码的开发中也被广泛采用，并且大部分的代码重用是未指明的；重用的代码在产品中能够占到 30%-90%；一些重要的软件，如 Linux 操作系统，GNU C 库，数据库软件，加密库和压缩库等，都广泛的存在代码重用；并且随着市场竞争，软件产品的复杂程度和质量的逐步提高，代码重用的趋势会进一步增强。

基于搜索的反编译器的具体技术受基于统计的机器翻译，基于搜索的软件工程，二进制代码源分析和二进制代码克隆检测等研究内容的影响。其核心是从二进制程序中索引相应的程序信息。

Rendezvous 系统采用 GNU C 库函数和 Linux 内核函数作为初始的语料库；采用指令助记符，控制流子图和数据常量作为索引二进制程序的 token 流。其中指令助记符被用来简单描述代码的语义，控制流子图被用来简单描述程序的结构，数据常量被用来简单描述数据的值。整个二进制程序的特征提取流程为，首先反汇编二进制程序中的所有过程体。然后，将每一个过程体进行分词，提取出指令助记符，控制流子图和数据常量等。最后，这些 token 经过特殊处理，被组合成索引字符串。整个系统的框架及其处理流程如下：

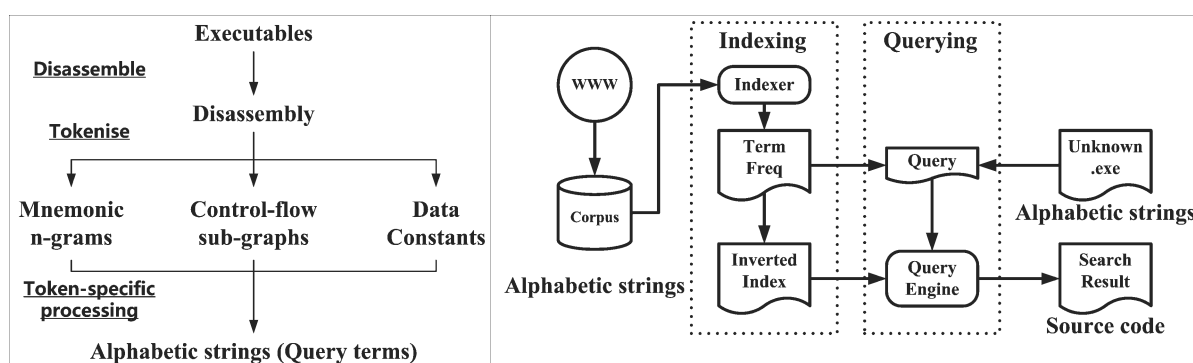


图 2-1 Rendezvous 系统工作流程

图 2-1 的左半部分展示的是 Rendezvous 系统的特征提取流程，右半部分展示的是 Rendezvous 系统的工作流程。被反编译的可执行程序，首先根据特征提取流程被转换成 Alphabetic strings，用作搜索引擎的输入；接着，Query 根据输入字符串到 Term Freq

模块去计算最佳匹配的词组序列；最后，Query Engine 根据最佳匹配的词组序列去 Inverted Index 中索引相应的源码，并将结果展示给用户。

Rendezvous 系统的语料库来自于网络中开源社区的软件，其构造过程如图 2-1 左半部分。实验结果表明，在 gcc 的-O1 和-O2 优化水平下，系统在 2.16 版本的 GNU C 库函数上能够达到 86.7% 的 F2 测度。在 gcc -O2 和 clang -O2 的优化水平下，系统在 6.10 版本的 coreutils 上能够实现 83.0% 的 F2 测度。

Rendezvous 系统在指令习语分析和结构化算法方面没有具体和特别的处理。

2.1.5 Hex-Rays 反编译器

Hex-Rays^[30]是工业界认可的反编译工具，是 IDA 反汇编工具的一个插件。Hex-Rays 的整个反编译流程包括微代码生成，局部优化，全局优化，局部变量分配，结构化分析，伪代码生成，伪代码变换，类型分析和变量重命名等九大步骤。从其优化组合来看，Hex-Rays 的扩展性等都不是非常良好。因此，目前 Hex-Rays 只支持 x86 和 arm 体系结构的反编译。

微代码生成包括函数的开场和结尾分析，switch 结构分析（实验测试表明，Hex-Rays 无法完全恢复基于平衡二叉树的 switch 结构）和函数整体验证；一条汇编指令通常被翻译成 5-15 条微指令。局部优化包括指令的简化，表达式的传播，基本块类型确定等，局部优化采用标准的窥孔优化算法。全局优化包括表达式的全局传播，死代码删除，分析被调用函数的参数和返回值；全局优化严重地依赖数据流分析算法。局部变量分配涉及到变量活跃范围分析以及其类型指派，同时要摆脱栈和寄存器引用。结构化分析从控制流图入手，分析出程序的高级控制结构（while/if/switch）。伪代码生成直接将蕴含有高级控制结构信息的微代码输出。伪代码变换通过创建 for 循环，break/continue 语句和删除多余的 goto 语句等方式，提高伪代码的可读性。类型分析在伪代码基础上建立类型方程组，计算并指派变量的类型。变量重命名通过给寄存器或者内存指定特定的名字，可以进一步提高伪代码的可读性。

Hex-Rays 在 x86 的指令习语方面的检测要好于 RD，Boomerang 和 REC^[99]等反编译器；在程序结构分析方面，采用结构化分析算法。

2.2 编译对程序结构的优化

1.1.3 节，介绍的编译优化主要集中在程序结构的优化方面。本节将从分支结构，循环结构和函数内联等方面来说明编译优化对程序结构的修改是普遍存在的。反编译所要处理的程序结构往往是这些编译优化遍的叠加效果。如同第 1.1.3 节所言，编译器的部分程序结构优化算法对于反编译结果的可读性是有益的，而部分程序结构优化算法对于反编译结果的可读性是无益的，这些程序结构优化算法是本节研究重点。

2.2.1 分支结构的优化

分支结构可以分为 if-then-else 结构和 switch 结构。对于 if-then-else 分支结构，当

其条件表达式为复合条件表达式时, 根据短路算法, 一个 if-then-else 分支结构会被转换成多个分支。If-then-else 结构的化简发生在 else 分支或 then 分支或二者均为空或者是条件表达式为常量的情况下。当 else 分支为空时, else 分支被删除; 当 then 分支为空时, 条件表达式被取反, 同时删除 then 分支; 当二者均为空时, 整个 if-then-else 分支都被删除; 当条件表达式为常量时, If-then-else 结构也可以被优化成单独的 then 分支或 else 分支取代。If-then-else 结构的化简可以被应用在反编译中, 简化程序结构。

Switch 结构对于理解程序的逻辑至关重要, 如果在程序解码过程中不能够有效地恢复 switch 结构, 会使程序的控制结构不完整, 后续的数据流分析, 类型推导和控制流分析不能正确地进行, 反编译结果无意义。

不同的编译器处理 switch 语句的方法各有所异, 即便是同一个编译器在处理不同类别的 switch 语句时, 方法^[31]也有所差别。基于搜索策略的方法将 switch 实现成一系列 if-equal 的分支嵌套语句, 嵌套的分支组成二叉树形状, 为了提高嵌套分支的执行效率, 编译器会根据 case 值的分布, 将整个嵌套分支的二叉树结构调整成近似平衡二叉树; 基于跳转表策略的方法将所有跳转偏移组成跳转表, 通过 case 值索引各个表项来实现 switch 语句; 基于组合策略的方法根据 case 值的分布, 对于不同段的跳转既可以采用基于搜索策略的处理方式也可以采用基于跳转表的处理方式。实验表明, Phoenix, Boomerang 等主流反编译器对于表现为嵌套的 if-equal 分支的 switch 语句, 没有进行特别的处理, 只是按照普通的结构化算法, 将其翻译为嵌套的 if-equal 分支。Hex-Rays 和 RD 对表现为嵌套的 if-equal 分支的 switch 语句进行过特殊处理, 但是处理的效果不是特别理想。如图 2-2 所示的反编译结果, 反编译在翻译时存在一些不足: Hex-Rays 只将 1001~1400 段的分支进行了有效的组合, 但是其它段的分支依然保留着嵌套的 if-equal 分支结构。具体的优化算法将在 3.2.2 节给出。

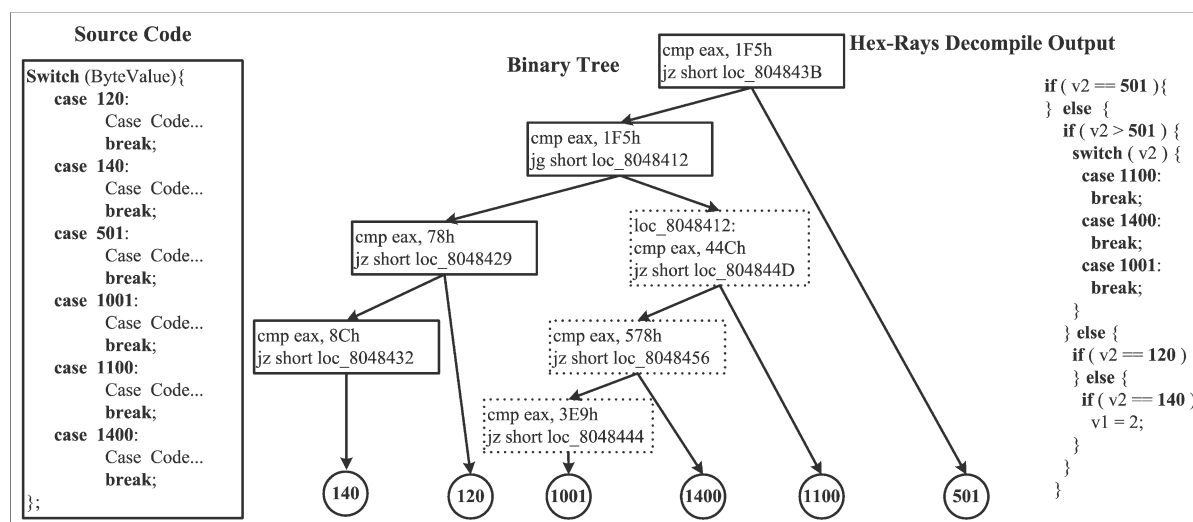


图 2-2 基于二叉树的 switch 结构翻译

根据 Hex-Rays 的技术特点, switch 分支的处理可能在后端的伪代码变换部分完成。但是, 最好在前端的控制流图上检测出相应的结构, 并将嵌套的 if-equal 分支转换成

N-way 结构。这样对于后续的反编译分析也是非常有益的。

2.2.2 循环结构的优化

根据 80-20 定律，程序执行的大部分时间花在少量的代码片段上面。循环作为程序中最耗时的部分，自然是编译优化的主要对象。如图 2-3 所示，loop unrolling 将原始控制流图中的 Head 结点和 Latch 结点复制了两遍，组成了一个体积更大的循环，降低了原始循环的迭代次数。Loop unrolling 经常和软件流水一起使用，增加循环迭代之间的并行度，提高循环的执行速度；Loop peeling 同样也将循环复制了两遍，但是被复制的代码不再是循环的一部分，而是被提出到循环外面，原始循环的循环次数被减少，通常用于优化迭代次数相对较少的循环。Loop inversion 将 while 循环转换成 repeat 循环，将循环前的循环结束测试语句移动到循环之后。Loop inversion 将循环结束时的指令判断次数减少到了一次。当循环中的分支表达式是循环不变量时，Loop unswitching 算法会将分支与循环的嵌套关系互换，即将原来的 while{if{}}else{}}结构转换成 if{while{}}else{while{}}结构。Loop unswitching 可以减少循环执行过程中的判断次数，从而加快循环的执行速度。

除上述具体优化算法外，程序中的 goto, break, continue 和异常处理等语句的优化，也会使原本规整的控制流图变得异常复杂。通常的结构化算法在分析此类结构时通常导致失败，无法获得很好的结果。

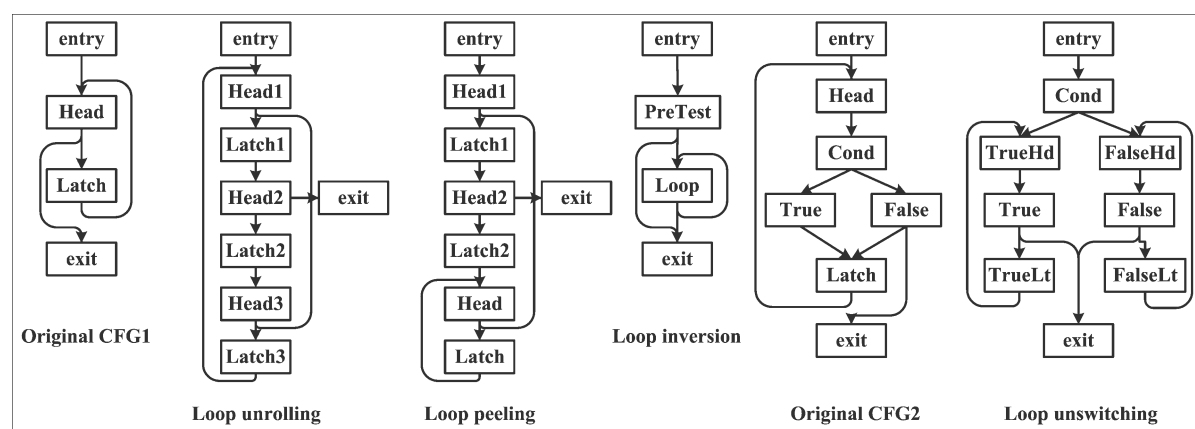


图 2-3 循环结构优化算法图示

图 2-3 展示的循环优化，结果不仅仅是基本块的数目增加，同时还使程序控制结构变得异常复杂。Loop unrolling 使得原来的单出口循环变成了多出口循环；Loop peeling 使得原来的循环外面嵌套了许多分支结构，增加了程序的嵌套深度，复杂化了程序的判断逻辑；Loop unswitching 使得程序的嵌套顺序改变，条件判断增多，程序逻辑变复杂。

总体上来看，编译优化算法往往使得原本结构良好的循环控制结构变得非常复杂。无形中增加了反编译研究中程序控制流图的结构化分析难度。

2.2.3 函数内联

函数内联是将被调用过程体在调用点处就地展开，从而消除调用开销，提高程序执行速度，同时降低软件开发难度的一种编译优化算法。在本文，函数内联包括固有函数的内联，宏定义展开和用户自定义函数的内联等。主流的编译器，如 GCC，LLVM 和 VC 编译器等都支持函数内联。

函数内联的好处如下：

首先，原本受函数调用而导致的寄存器分配，公共子表达式消除，指令调度，代码压缩和常量传播等算法失效。在函数内联之后，这些优化算法的优化范围进一步扩大。

其次，将循环中的函数调用内联展开，可以提高循环的向量化和执行并行度。

最后，内联函数在软件工程中具有重要作用，因为内联函数的存在可以降低软件开发的复杂度^[32]。

由于函数内联会增加代码的体积和增加指令缓存失效的概率，因此函数内联是一个优化问题。函数内联的一般步骤如下：

(1) 确定在编译的哪个阶段进行函数内联。函数内联展开既可以在编译阶段进行也可以在链接阶段进行。在编译阶段，程序的高级结构信息，如循环和分支等被保留，利于上下文分析。但是在编译阶段进行函数内联展开的不足之处是，由于单独编译的原因，许多被展开的函数在进行内联展开之前还没有被编译，不能掌握其有效特征信息。在链接阶段，所有过程的信息都存在，可以更好的展开分析。但是，在展开之后，还需要对代码进行优化分析。

(2) 选取所有的内联函数展开点。内联函数展开点的确认受到代码体积大小和控制栈大小的双重约束。通常通过代价函数来评测具体的内联函数展开点。

(3) 内联函数展开。展开过程包括复制被调用函数，变量重命名和符号表的修改等。

现在的主流编译器在进行函数内联时，都采用具体的启发式算法或机器学习算法^[102,103]，从而更好的确定函数在其调用点处是否被展开。

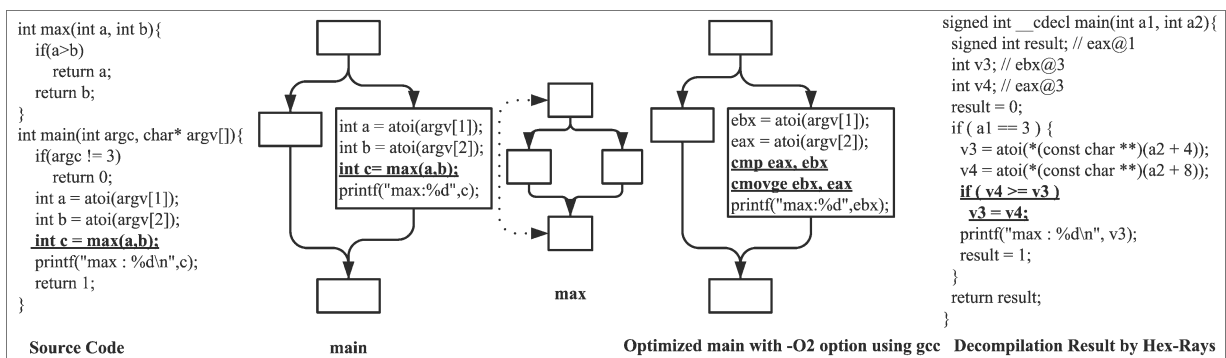


图 2-4 函数内联及其反编译结果示意

图 2-4 展示了 max 程序源码，main 过程体和 max 过程体的精简和抽象的控制流图，在 GCC 编译器的-O2 优化之后的整个程序的控制流图以及 Hex-Rays 的反编译结果。

从中可以看出，max 函数的函数调用栈及其相关程序代码被转换成条件移动指令 `cmp` 和 `cmovge` 的组合。Hex-Rays 将这两条指令的组合翻译成了一个 if-then 结构。

函数内联对于程序执行来说是有益的，因为其可以提高程序的执行速度。但是，函数内联对于反编译来说，无形中增加了反编译结果的代码量，降低了反编译结果的抽象程度，增加了逆向分析的复杂度。因此，在反编译中研究内联函数的识别和抽象是非常有意义的。3.4 节将就此展开进一步的说明。

除此之外，代码迷乱技术通过增加机器代码的控制流的复杂度，提升逆向分析中的静态分析和动态分析的难度，从而达到软件保护的目的。文献^[33]使用 Petri 网来迷乱程序的控制流图，从而达到软件保护的目的。文献^[34]提出一种两进程迷乱程序控制流图的机制，P-process 中的程序在热结点边界处被随机排序，而排序本身来自另外一个进程 M-process 的跳转表中。当 P-process 在执行过程中遇到排序指令时，P-process 会向 M-process 发送地址转换查询请求。文献^[35]通过将分支指令转换成异常处理代码并在异常处理代码之后插入伪造分支指令的方式，达到迷乱程序控制流图的目的。

2.3 反编译中的结构优化

程序的结构对于理解程序至关重要，程序中最符合人的思维方式的程序有顺序结构，分支结构和循环结构等高级语言中的常用结构。程序结构优化可以提高后续反编译中数据流分析和控制流分析的效率与精度。反编译中的程序结构优化的核心目标是从复杂和非规整的控制结构中，恢复出简洁的高级控制结构。

2.3.1 Goto 语句优化

Goto 语句属于非结构化编程，在现在的高级语言中已经基本被舍弃，以降低代码维护的复杂度等。反编译的处理对象通常是高度优化的程序，程序的结构高度精简，致使反编译中的结构化算法不能胜任。通常，在结构化算法不能分析程序的结构时，会引入 goto 语句。根据文献^[36]，程序的非结构化因素经常由不规则的选择路径，多出口循环，多入口循环，重叠循环和并置循环等五大因素引起。图 2-5 中的 Abnormal selection path 展示了不规则的分支结构，Loop with multiple exits 展示了有多个出口的循环，Loop with multiple entries 展示了有多个入口的循环，Overlapping loops 展示了两个重叠的循环，Parallel loops 展示了两个并置的循环。文献^[36]证明了所有的程序中的不可分解结构都含有图 2-5 中的五种常见结构，并给出了从六种复杂的控制结构里分解出图 2-5 中五种非结构化控制流子图的过程。

Goto 语句的优化有多种方法，包括引入 Boolean 变量，代码复制，多出口循环的使用和常用语言中不存在的高级结构的使用等。

文献^[37]在 AST (Abstract Syntax Tree) 上，首先进行 goto 语句移动变换，接着对部分 goto 语句进行适当的删除操作，进而达到消除 goto 语句的目的。在文中，作者给出了 goto 外提，goto 内移以及 goto 提升等三种方法来实现 goto 语句的移动，进而为消除提供依据；同时，提出了通过变换条件表达式消除 goto 语句和循环代替 goto 语句

等两种 goto 语句消除算法。在整个过程中要不断地引入新的条件变量，以保证程序语义的正确性。

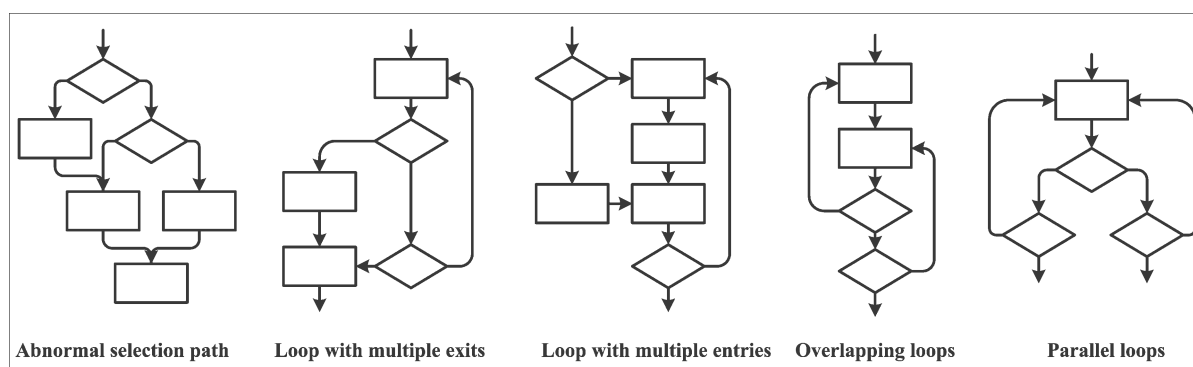


图 2-5 非结构化控制流程图

文献^[38]将控制流图建模成正则表达式，结构化的控制流图被转换成结构化正则表达式 sre (Structured Regular Expression)；非结构化的控制流图被转换成非结构化正则表达式 buf (Basic Unstructured Form)。将控制流图的正则化看成是不断的在正则表达式中发现 buf 并将其替换为相应的 sre 的迭代过程。在整个过程中，控制节点被不断的复制，从而保证程序逻辑的正确性。

文献^[39]首先介绍了 GOTO 语言和 EXIT 语言的区别，然后，通过将 GOTO 程序翻译为 EXIT 程序的方法，实现 goto 语句的删除。

以上都是编译领域中的相关研究，但是这些算法的思想同样可以被用到反编译领域，用于消除反编译过程体中的 goto 语句。但是，消除 goto 语句的根本应当从提高控制流图的结构化分析入手。通常，goto 语句的引入是因为对目标程序的结构分析不充分而引起的，例如在程序的结构化算法中有顺序，分支和循环的分析，但是很少有 break 和 continue 结构的分析。

2.3.2 控制流图的结构化

控制流图的结构化完成程序中基本块的分类，为后续代码生成提供高级控制结构信息。最早的控制流的结构化分析是基于模式识别的，文献^[40]提出一种时间复杂度为 $O(N)$ 的良构变换算法，其核心是构造了 13 个变换法则用于将低级的控制结构转换成良构的高级语言控制结构。然后在程序控制流图中迭代检测所有低级的控制结构，并不断地应用 13 个变换法则完成低级结构到高级结构的替换，直至低级的程序控制流图变成结构良好的高级控制结构。在变换过程中，当没有变换规则可以使用时，控制边被删除，并在适当的位置产生 goto 语句。目前，常用的控制流的结构化分析分为两种，分别是基于控制流图的区间分析 IA (Interval Analysis) 的算法^[41]和基于结构化分析 SA (Structural Analysis) 的算法^[42]。

经典的控制流的结构化分析^[43]都是基于对控制流图的区间分析 IA，从原始控制流图出发推导出一系列图序列，并对图序列中的每个图展开区间分析 IA。其中，区间信息中蕴含着顺序，分支和循环等基本结构信息，而图序列中蕴含着程序中各种控制结

构的嵌套关系。结构化分析 SA 被广泛的应用在 DCC, Boomerang, Phoenix 和 Hex-Rays 等反编译器中。Hex-Rays 反编译器通过结构化分析 SA 识别程序中的高级结构信息, 并分析控制流图中的嵌套关系, 构建出程序的控制树结构^[44], 然后根据控制树生成初始化的伪代码。

文献^[45]给出了在控制流图上检测复合条件表达式控制流子图以及级联条件子图模式, 分别用来组合复合条件判断语句中的各个条件表达式以及抽象 switch 结构。但是文中只给出了具体的检测算法, 没有给出如何恢复这些结构的算法, 具体算法将在 3.2.2 节给出。文献^[46]通过深度挖掘控制流图的深度优先遍历中所产生的前向边, 交错边, 回边以及额外的四大性质, 最终提出了一种 $O(N)$ 级的循环检测算法。实验表明, 算法要比 Havlak-Tarjan 算法快 2-5 倍, 比 Ramalingam-Havlak-Tarjan 算法快 2-8 倍。

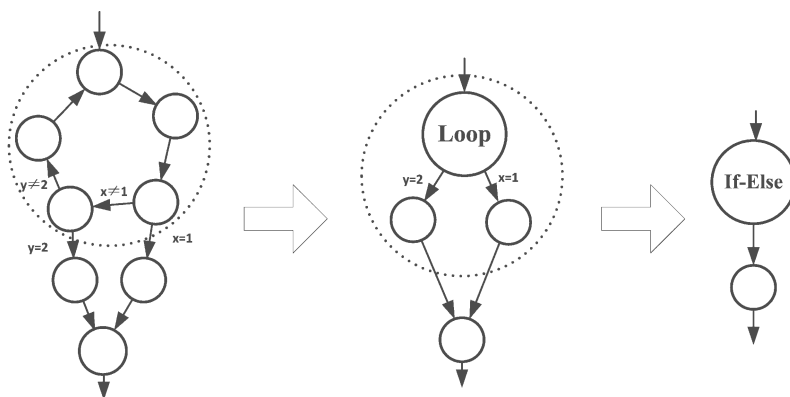


图 2-6 不保持语义的程序结构化过程

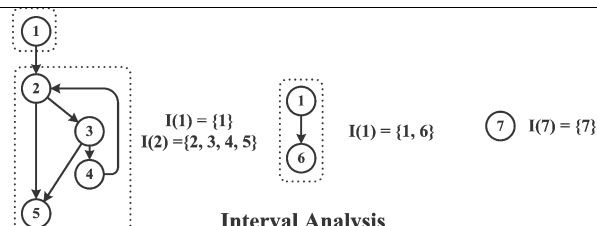
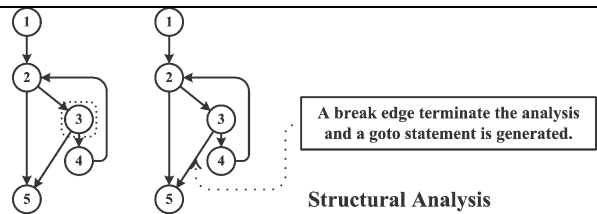
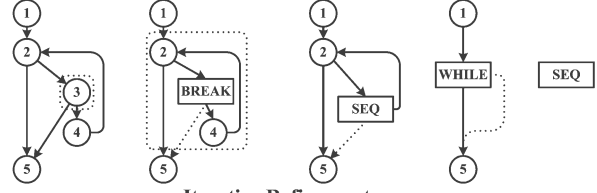
但是, 现有的结构化分析 SA 在某些情况下是不能保证程序语义的, 同时其分析粒度比较粗糙。其示例如图 2-6, 首先, 虚线圆圈中的基本块被识别为多出口循环结构; 接着, If-Else 分支结构又被结构化分析 SA 识别出来; 最后, 程序被很好的结构化。但是, 在分析 If-Else 结构时, 由于没有考虑分支条件, 导致构造出来的分支具有不确定性, 即当 $x==1$ 和 $y==2$ 同时成立时, 程序的两个分支都将被执行, 因而违背了程序执行的确定性。因此, 文献^[26]提出了一种既保证语义, 且迭代式分析程序中的控制结构的结构化分析 SA 算法, 迭代式精化 IR 算法。区间分析 IA, 结构化分析 SA 和迭代式精化 IR 对比如表 2-1。

通过表 2-1 分析可得, 提高结构化算法应当从以下三个方面着手:

- (1) 采用比 Interval 更小的分析范围。
- (2) 设计更多的匹配规则, 可以处理图 2-5 的非结构化控制流子图。
- (3) 迭代式地分析和精化程序结构。

文献^[47]介绍了一种名叫模式独立的结构化算法, 可以完全消除反编译结果中的 goto 语句。根据实验, DREAM 反编译器所采用的结构化算法消除了 coreutils 测试集上的所有 goto 语句。该算法具体包括两大部分, 第一部分是关于控制流图的结构化, 控制流图经过模式独立和语义保持的变换处理后, 被转换成抽象语法树; 第二部分是关于抽象语法树的优化, 抽象语法树经过控制结构化简, 内联字符串处理函数的抽象和变量重命名之后被转换成可读性更高的抽象语法树。

表 2-1 三种结构化算法特点分析对比

算法处理流程图示	算法特点分析
 <p>Interval Analysis</p>	<ol style="list-style-type: none"> 1. 从程序控制树的角度来看，是一种自底向上的规约方法。 2. 区间通常比较大，分析粒度比较粗。 3. 基于规则匹配的方法，对于不符合规则的结构，生成 goto 语句代替。
 <p>Structural Analysis</p>	<ol style="list-style-type: none"> 1. 从程序控制树的角度来看，是一种自底向上的规约方法。 2. 基于规则匹配的方法，对于不符合规则的结构，生成 goto 语句代替。
 <p>Iterative Refinement</p>	<ol style="list-style-type: none"> 1. 从程序控制树的角度来看，是一种自底向上的规约方法。 2. 对于不符合规则的结构，采取迭代式逐步精化的过程进行结构化。

DREAM 反编译器不仅能消除反编译结果中的 goto 语句，同时还可以产生更加紧凑的代码。实验结果表明，DREAM 反编译器产生的代码量是 Hex-Rays 的 72.7%，是 Phoenix 的 98.8%。图 2-7 展示了 DREAM 反编译器中的结构化算法的流程示意以及反编译结果与 Hex-Rays 反编译器的对比结果。

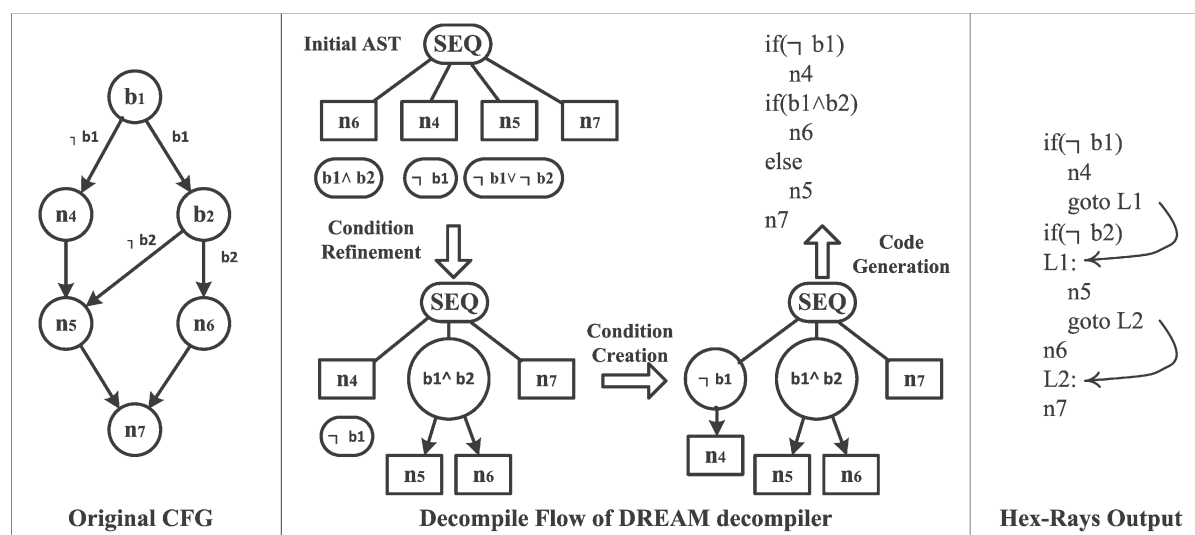


图 2-7 DREAM 反编译器与 Hex-Rays 反编译器对比

图 2-7 最左边展示的是原始的控制流图，该控制流图由六个基本块构成，是典型的复合分支结构在短路算法优化之后的结果。该程序在 Hex-Rays 反编译器中的反编译结果如图 2-7 最右边所示，在程序中存在两个 goto 语句以及两个标号。该程序在 DREAM 反编译器中的反编译流程如中间部分所示。原始的控制流图经过结构化算法

之后被转换成初始的 AST 结构，其条件表达式被标注在下方；接着，AST 经过条件表达式精化之后被转换成左下方的 AST；再经过条件表达式创建之后，AST 变成右下方所示的结构；最后，通过代码生成步骤，生成结构良好的 C 程序。通过对比发现，在程序结构方面，DREAM 反编译器的结果要明显好于 Hex-Rays 反编译器的结果。

2.4 图论的相关研究

图是用来建模物体之间相互关系的数学结构，而图论是研究图的特性的学科。在编译器和反编译器的优化算法设计中，经常用图来建模其中的关系，并利用图论知识研究相应领域的问题。文献^[48]总结了控制流图的性质，可约解控制流图的性质以及图算法在数据流分析中的应用，说明了图论在编译优化算法中的广泛使用。在反编译^[15]的数据流分析，类型分析和控制流分析中，也需要利用图对其中的关系建模。文献^[49]和文献^[50]分别利用 Similarity Flooding 算法^[51]和基于图核的机器学习算法对基于控制流图表示的程序进行聚类，从而发现程序中相似的代码模式，控制流和上下文，为程序的深度优化（例如对于相似的过程体采取相同的编译优化遍和优化组合）提供依据。文献^[52]对 180 多篇论文进行研究，总结了图论技术在模式识别中的广泛应用，并对图匹配，图嵌入，图核函数，图聚类和图学习等问题进行了深入探讨。文献^[53]提出了一种用于评价 CFG 相似性算法有效性的方法，该方法首先根据 CFG 样本，生成特定编辑距离的 CFG 集合。然后，利用各种不同的 CFG 相似性求解算法计算 CFG 样本和 CFG 集合中的 CFG 的相似性。最终，通过打分的方式判断各个 CFG 相似性算法的优劣。下面两节将主要从图同构和图拓扑嵌入两方面来研究图论知识在编译和反编译领域的应用。

2.4.1 图同构

图同构是研究两个图的顶点之间是否存在双射关系的图论研究内容。形式化的定义如下：

图 $G_1=(V_1, E_1)$ 同构于图 $G_2=(V_2, E_2)$ 当且仅当存在一个双射函数 $\Phi:V_1 \rightarrow V_2$ ，使得对于 G_1 中的任意两个顶点 $v_i, v_j \in V_1$ ，且 $(v_i, v_j) \in E_1$ 当且仅当 $(\Phi(v_i), \Phi(v_j)) \in E_2$ 。

图同构中最重要的问题是子图同构，其形式化定义如下：

图 $G_1=(V_1, E_1)$ 子图同构于图 $G_2=(V_2, E_2)$ 当且仅当存在 G_2 的一个子图 $G_3=(V_3, E_3)$ ，且存在一个双射函数 $\Phi:V_1 \rightarrow V_3$ ，使得对于 G_1 中的任意两个顶点 $v_i, v_j \in V_1$ ，且 $(v_i, v_j) \in E_1$ 当且仅当 $(\Phi(v_i), \Phi(v_j)) \in E_3$ 。

子图同构问题已经被证明是一个 NP 完全问题，其时间复杂度随着顶点数的增加而呈指数级增长。在一些情况下，由于顶点的信息非常重要，因此在计算子图同构时还要考虑顶点的语义。因此，子图语义同构形式化定义如下：

图 $G_1=(V_1, E_1)$ 子图语义同构于图 $G_2=(V_2, E_2)$ 当且仅当存在 G_2 的一个子图 $G_3=(V_3, E_3)$ ，且存在一个双射函数 $\Phi:V_1 \rightarrow V_3$ ，两个顶点语义映射函数 $\Psi:V_1 \rightarrow L$ 和 $\Psi':V_3 \rightarrow L$ ，使得对于 G_1 中的任意两个顶点 $v_i, v_j \in V_1$ ，且 $(v_i, v_j) \in E_1$ 当且仅当 $(\Phi(v_i),$

$\Phi(v_j) \in E_3$, $\Psi(v_i)$ 与 $\Psi'(\Phi(v_i))$ 的语义相兼容, $\Psi(v_j)$ 与 $\Psi'(\Phi(v_j))$ 的语义相兼容。

文献^[54]给出了在 GCC 编译器上进行自动化的指令集扩展和复杂指令选择的方法。复杂指令选择利用了图同构的方法,使得 GCC 编译器能够自动开发和重用复杂指令模式,获得比现有方法更加可扩展和强大的 ISE (Instruction Set Extension) 方法。

文献^[55]首先将可执行程序 P 进行反汇编,得到程序 P';接着,对程序 P'进行规范化操作以消除自我变异恶意代码中的突变技术 (mutation techniques) 和发现正常代码和恶意代码之间的控制关系,得到程序 PN;再接着,在程序 PN 的基础上建立带有标号的过程体间控制流图 CFGPN;最后,将正规化的恶意代码 CFGM 与 CFGPN 进行对比,确定 CFGM 是否子图同构于 CFGPN。

文献^[56]通过判断源程序的控制流图, CoSy 编译器中间表示的控制流图以及 CoSy 编译器产生的目标汇编代码的控制流图的同构性,来验证 CoSy 编译器的安全性,并确定不安全因素的位置。

最早的子图同构算法 Ullmann 算法^[57],采用带回溯的树搜索技术。VF 算法^[58]和 VF2 算法^[59]在树搜索技术中引入启发式规则对搜索树进行剪枝,在性能上都要优于 Ullmann 算法。目前,学术界公认的最快的图同构判定算法是 Nauty 算法^[60]。

经过以上分析,可以看出图同构在指令选择,恶意代码分析和编译器验证等领域有着广泛的应用,并且图同构算法研究比较深入。在反编译领域,同样也存在着许多和图相关的模式,具体内容将在 3.1.1 节和 3.4 节介绍。3.1.1 节主要介绍在基本块内,抽象指令的语义;3.4 节从固有函数的识别和抽象来说明子图语义同构的应用之处。

2.4.2 图拓扑嵌入

图同构是一种精确的图匹配算法,而图拓扑嵌入就相对比较宽松,它不要求保证严格的父亲关系,而只需保证祖先关系即可。在介绍图拓扑嵌入的定义之前,先约定一些记号。 $r(G)$ 表示图 G 的根节点, $l(x)$ 表示结点 x 的属性, $d(x)$ 表示结点 x 的度数, $sub(x)$ 表示根节点为 x 的子图, $ci(x)$ 表示结点 x 的第 i 个孩子结点, $subtree(x)$ 表示根节点为 x 的子树。图拓扑嵌入的形式化递归定义^[61]如下:

图 P 拓扑嵌入到图 T 中,满足如下三个条件:

- (1) $l(r(P)) = l(r(T))$, 即图 P 的根节点的标号与图 T 的根节点的标号相匹配。
- (2) $d(r(P)) = 0$ 或者 $d(r(P)) = d(r(T))$ 且满足 (3)。即图 P 的根节点的度为 0 或者图 P 的根节点的度数与图 T 的根节点的度数相等,且满足 (3)。
- (3) 对于由 $r(P)$ 的每一个孩子结点为根展开的子树 $subtree(ci(r(P)))$, 都要拓扑嵌入到 $subtree(di)$ 中,其中 di 或者是 $r(T)$ 的孩子 $ci(r(T))$ 或者是 $ci(r(T))$ 的后代。

文献^[62]从研究编译器中的指令习语识别入手,分析了程序在经过不同的编译优化算法处理后,其结构和数据流方面的巨大变化,导致传统的精确匹配的方法失效,错失指令习语的优化的可能。作者提出了基于拓扑嵌入检测程序中的指令习语的方法。该方法遍历程序中的每一个内层循环,并将循环转换成控制流图 T,再遍历每一个指令习语控制流图 P。如果 P 中的所有结点都出现在 T 中且循环迭代次数足够多,则检

测 P 是否拓扑嵌入到 T 中；如果是，则创建 UD/DU 链，执行 loop peeling 算法，复制存储结点，代码移动。最后，如果 T 和 P 相匹配了，则将该循环用事先定义好的，效率更高的代码段替换。实验结果表明，在 Java Compatibility Kit API 测试集上，该算法比基于精确匹配的算法多转换 76% 的循环。在性能方面，在 IBM XML parser 测试集上，可以提升 54%，在 SPECjvm98 和 SPECjbb2000 上分别获得 1.9% 和 4.4% 的性能提升。而整个即时编译所花费的时间只从原来的 0.32% 增加到了 0.44%。

专利^[63]利用拓扑嵌入的方法来度量待处理过程体与数据库中组代表元的相似度，进而将该过程体进行分组，以辅助程序分析。

文献^[64]对比分析了基于图核与基于图嵌入的图分类算法，并指出基于图嵌入的图分类算法在性能上可以与基于图核的算法相媲美。同时，由于图嵌入在表达方面要优于基于图核的机器学习算法。因此，基于图嵌入的图分类算法是一个非常具有前途的技术。

2.5 本章小结

本章先介绍了当前具有代表性的反编译器及其关键技术，并分析它们在软件框架方面的特点和不足，为 ASMBloom 反编译器框架的设计（3.1 节）奠定基础；研究反编译器在程序结构优化方面所采用的优化算法。

2.2 节研究编译对程序中的主要结构（分支，循环和函数内联）的优化算法和程序迷乱技术，说明了反编译过程中程序结构优化所面临的严峻挑战；同时，为反编译器中的结构优化算法提供技术思路。

2.3 节研究反编译中程序结构优化的主要挑战——goto 语句优化和控制流图的结构化。通过研究编译器中对 goto 语句优化算法，借鉴思路，为反编译器中的 goto 语句消除（3.3 节）提供技术支持。通过对比分析三种结构化算法——区间分析 IA，结构化分析 SA 和迭代式精化 IR，说明各自的特点以及今后结构化算法改进的方向。

最后，研究图论中的图同构和图拓扑嵌入问题。说明这些算法在编译器和反编译器优化算法中的普遍应用。图同构和图拓扑嵌入的研究，为 3.4 节提供技术理论支持。

3 程序结构优化

本章先介绍反编译器的框架，再介绍一些具体的反编译优化算法。反编译器的框架主要从智能解码器和中间表达式着手，说明 ASMBoom 反编译器是一个智能的可扩展的系统。优化算法主要集中在 switch 结构优化，goto 语句消除以及内联固有函数消除等三方面。

3.1 反编译器框架

通过 2.1 节对各种主流反编译器的总结对比以及分析，在反编译器框架的设计方面应当遵从如下几个原则：

- (1) 提高 IR 的抽象程度。
- (2) 提高 IR 的通用性和兼容性。
- (3) 扩展性良好。

原则（1）要求在生成具体 IR 时，应当先进行指令习语的检测：消除与机器操作密切相关的操作，封装部分内联库函数和指令习语，简化后续分析复杂度。这也是 3.1.1 节讲述智能解码器的原因。原则（2）中的通用性要求反编译器的 IR 最好是主流编译器所采用的中间代码，这样有助于利用编译中的优化算法解决反编译中的问题，如数据流分析，常量传播，复制传播等。原则（2）中的兼容性要求反编译器的解码器能够生成多种 IR，以方便不同反编译器之间的对比，分析各自解码器的差异以及后续优化算法和优化算法组合的优劣。原则（3）要求反编译器不仅能够从二进制分析生成高级代码，由于逆向工程的分析范围非常广阔，因此需要反编译器的扩展能力比较好，即系统的功能模块化水平比较高，容易实现功能模块的增删改。

在满足以上三个原则的情况下，最终的 ASMBoom 反编译器框架如下：

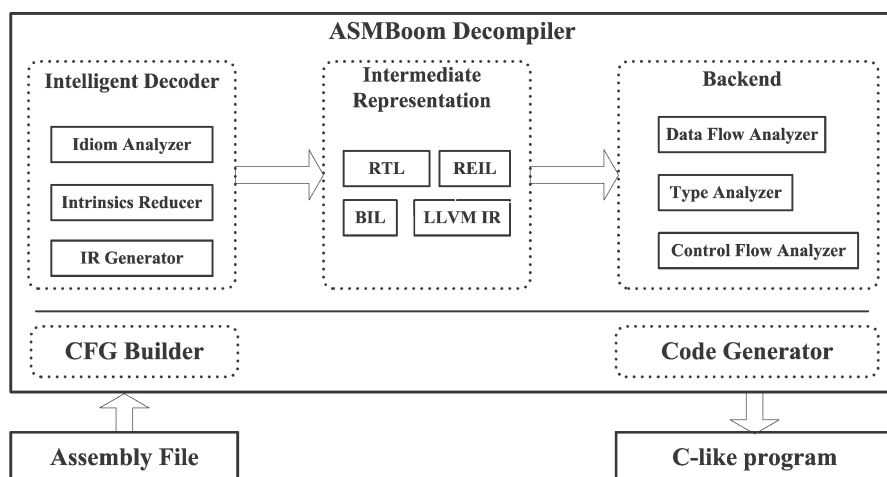


图 3-1 反编译器整体框架

反编译器的输入为汇编文件，可以是 Hex-Rays 的反汇编输出。反编译器的输出为类 C 代码。在汇编代码输入之后，CFG Builder 根据控制流图构建算法^[65]，从线性的汇编指令流序列中创建程序的控制结构，创建过程中包括 switch 结构的恢复，将在 3.2 节讨论。紧接着基于控制流图的汇编程序并没有直接被转换成中间表达式 IR，而是经过 Intelligent Decoder（智能解码器）的预处理才生成中间表达式 IR。Intelligent Decoder 包括三部分：Idiom Analyzer，Intrinsics Reducer 和 IR Generator。Idiom Analyzer 利用类似于窥孔优化的算法分析基本块内的指令习语，并将指令习语转换成抽象程度更高的操作；Intrinsics Reducer 根据子图同构算法识别程序控制流图中的内联固有函数，分析其参数和返回值，删除原有控制流子图，并将其抽象为函数调用指令。IR Generator 负责中间代码的生成，其具体生成算法将在 3.1.2 节给出。Intelligent Decoder 的提出，可以提高 IR 的抽象程度，满足了原则（1）。

Intermediate Representation 模块展示了 RTL, REIL (Reverse Engineering Intermediate Language)^[66]，BIL (Binary Intermediate Language)^[4]和 LLVM IR 等反编译领域的四大主流中间表达式，其具体细节及对比将在 3.1.2 节给出。支持多种中间语言的中间端可以增加反编译器的通用性和兼容性，符合原则（2）。

反编译器的后端 Backend 由 Data Flow Analyzer，Type Analyzer 和 Control Flow Analyzer 等三大部分组成。Data Flow Analyzer 负责数据流分析^[67]，包括活跃变量分析，到达定值分析，函数参数和返回值分析，常量传播和复制传播等。Type Analyzer 负责程序中变量的类型分析^[68]，其原理是从机器指令的操作码，库函数签名和常数等多处获取基本类型信息，然后利用类型推导规则推导其它变量的基本类型，从而使得生成的高级代码的可读性更强。Control Flow Analyzer 根据结构化算法^[41]，将控制流图中的结点按照其在控制流图中的位置分成不同的类别，如顺序控制流子图、分支控制流子图和循环控制流子图等。后端的高度模块化增加了反编译器的可扩展性，符合原则（3）。

最后，高度结构化的程序经过 Code Generator，将中间代码转换成高级的类 C 代码。

3.1.1 智能解码器

智能解码器是在研究众多反编译器的解码器设计思路的基础上，根据实际经验提出的一种新型解码器。传统的解码器的局限性在于，在解码过程中，只关注单独的一条指令——取操作码，操作数；在中间语言映射词典中查找其对应的中间表达式；参数替换。根据以上三步，就完成了指令向中间代码的转换。

传统的解码器完全忽略了指令的上下文环境，这种不分析指令的上下文环境的机械地解码往往不能取得很好的效果。根据 RD 反编译器的实验结果，在 x86 程序的指令习语分析中，RD 的检出率只有 21.6%，明显低于 Hex-Rays 的 63.5% 的检出率。其本质原因是 RD 的指令习语分析在 LLVM IR 中间表达式上进行，再加上 x86 指令系统的语义复杂以及 LLVM IR 的描述过长等原因，最终造成了 RD 的指令习语的检出率较低。

智能解码器在基于控制流图的汇编程序上进行指令习语检测和内联固有函数消除，其优势分析如下：

(1) 在汇编上检测指令习语可以提高 IR 的通用性。因为指令习语在不同的 ISA 上表现不同，因此需要为不同的 ISA 构造不同的 IR 指令习语模板。

(2) 在基于控制流图的汇编程序上进行习语分析，可以消减 IR 的数目和降低控制流的复杂度，进而可以有效简化后续数据流分析和控制流分析。

(3) 在基本块内进行指令习语检测，使得指令习语分析有了边界，提高了指令习语分析的精确度。

(4) 尽早的抽象内联固有函数，能够为后续分析提供更丰富的类型信息，有助于类型分析。

指令习语的分析采用类似于窥孔优化^[69, 70]的算法来检测指令序列流中的习语，如果检测到，则按照指令习语解码；否则按照普通指令解码。但是，由于反编译的输入是高度优化的代码，尤其是指令调度^[71]对于指令习语的影响也非常大，因此在进行窥孔优化之前，必须对基本块中的指令进行聚类，其聚类的依据是指令之间的定义-使用关系，根据定义-使用关系将基本块中的指令分成不同的类别，在编译中常见的方法是程序切片^[72]。

算法 3-1 的复杂度为 $O(N^2)$ ，其中 N 为过程体中指令的数目。

分析过程：设过程体中有 K 个基本块，第 i 个基本块中的指令个数为 s_i ，则有 $N = \sum_{i=1}^K s_i$ 。算法每次要遍历每个基本块中的指令 s_i^2 次，因此总的时间复杂度为 $\sum_{i=1}^K s_i^2$ ，由于 $\sum_{i=1}^K s_i^2 \leq \left\{ \sum_{i=1}^K s_i \right\}^2 = N^2$ 。因此，算法的时间复杂度为 $O(N^2)$ 。

算法的流程如算法 3-1 所示，算法遍历程序控制流图中的每一个基本块，对于基本块中的指令执行两步操作。第一步根据指令之间的定义-使用，求取基本块中指令的程序切片。其具体过程为，首先根据定义-使用关系构建指令之间的依赖图；然后求取依赖图中的所有连通分支，其中的每一个连通分支代表一个程序切片。第二步遍历每一个程序切片，并利用窥孔优化算法，在切片中匹配有效的指令习语，最后完成指令习语的抽象及其参数的替换。

Proc indiom_Lift()

/*Parameter and Return Illustration*/

Input: an assembly procedure P expressed in Control Flow Graph(CFG) format.

Output: an assembly procedure with Instruction Idiom reduced and abstracted.

0. /*Traverse each basic block of the procedure*/
 1. **for** each basic block B in procedure P's CFG:
 2. /***Step 1** Cluster instructions in B using program slicing*/
 3. /***step 1.1** Build the data dependence graph*/
 4. depGraph = NULL
 5. **for** each instruction Ix in B
 6. Ix_defs = defines of instruction Ix
 7. **if** Ix not visited
 8. Insert node Ix into depGraph and Mark Ix visited
-

```

9.      end if
10.     for each instruction Iy after Ix in B
11.         Iy_uses = uses of instruction Iy
12.         if Ix_defs  $\cap$  Iy_uses  $\neq \emptyset$ 
13.             if Iy not visited
14.                 Insert node Iy into depGraph and Mark Iy visited
15.             end if
16.             Insert edge  $\langle Ix, Iy \rangle$  into depGraph
17.         end if
18.     end for
19. end for
20. /*Step 1.2 Compute the slices(each connected component represents a slice)*/
21. slices = connected_Components(depGraph)
22. /*Step 2 Peephole optimize the slices*/
23. for each slice S in slices
24.     /*Repeat until no pattern found; pattern organized in descending order in Pm*/
25.     repeat
26.         for each pattern Pt in pattern map Pm
27.             if  $Pt \subseteq S$ 
28.                 Replace the instructions in S with the right side of Pt
29.                 Substitute the arguments
30.             end if
31.         end for
32.     until no pattern matched
33. end for
34. end for
end Proc

```

算法 3-1 指令习语提升算法

算法 3-1 中最复杂的部分是指令习语模板库的构建,传统的做法是利用人工实现,其步骤为:首先,将包含相应指令习语的源码 *S* 用相应的编译器优化编译,得到可执行程序 *E*。接着,利用反汇编工具反汇编程程序 *E*,得到反汇编结果 *D*。最后,对比源码 *S* 和反汇编结果 *D*,分析源码 *S* 中的高级操作 *H* 与反汇编结果 *D* 中的低级操作 *L* 的对应关系,找到其中的 $\langle H, L \rangle$ 键值对,并将这些键值对存储在模板库中,以供指令习语分析使用。

这样的工作是繁琐而机械的,可以利用数据挖掘中的方法代替,例如在构建指令习语是,可以利用频繁顺序模式挖掘算法^[73, 74]找到程序的一个基本块中的程序切片中的所有频繁顺序项集,将这些频繁顺序项集作为指令习语,并抽象其操作。因为,在真实的反编译环境中,面临的程序往往是源码未知,编译器未知,编译优化算法未知等。利用数据挖掘的方法,可以加快分析。文献^[75]利用顺序模式挖掘算法分析硬件取样数据,新方法明显地加快了硬件取样数据的分析速度。实验结果表明,近 50%的频繁模式跨越基本块。但是,这并不影响我们在一个基本块的切片内挖掘指令习语,因为硬件取样数据是程序动态运行的结果,又由于循环在程序的执行过程中占很大比例,因此其 50%的频繁模式跨越基本块很正常。

关于 switch 结构的恢复和规整以及在基于控制流图的汇编程序上消减内联的固有函数的相关算法，将分别在 3.2 节和 3.4 节展开介绍。

3.1.2 中间表示生成

图 3-1 列出了逆向工程领域常见的中间表达式。其中 RTL 是 Boomerang 反编译器，Jakstab 反汇编器和 UQBT^[76] 二进制翻译器的中间语言。

LLVM IR 是 LLVM 编译器的中间语言，该语言是摆脱了物理寄存器，流水线和函数调用约定等限制的抽象指令集，该指令集中包含 31 个操作码；LLVM IR 是 SSA 形式的，很大程度上简化了数据流优化；LLVM IR 采用了独立于任何语言的类型系统，只包括 void, bool, 长度从 8 到 64 位的有符号整数和无符号整数，单精度浮点类型和双精度浮点类型等基本类型，以及指针，数组，结构体，函数和 SIMD 向量等派生类型。LLVM IR 的内存访问采用统一的 load/store 完成，栈空间分配使用 alloca 指令，堆空间管理采用 malloc 和 free 指令。这些特点使得 LLVM IR 在各个领域获得广泛使用，在编译领域，Haskell^[77]，Scheme，Scala^[78] 等语言的中间分析都在转向 LLVM 中间表达式；在二进制翻译方面，LLVM-QEMU^[79] 通过将 x86 可执行程序翻译成 LLVM IR，然后利用 LLVM 中的优化遍优化代码，最后通过即时编译器重新生成 x86 可执行程序。这样的翻译可以被用来跨平台运行，例如即时编译器可以重定向生成其它体系结构的可执行程序；在逆向分析方面，Dagger 是建立在 LLVM 基础上的反编译框架，用于将目标指令翻译为 LLVM IR，可以被用在二进制重写，二进制翻译，指令集模拟和反编译等领域。

BIL 是 BAP 项目的中间语言，该中间语言由赋值表达式，跳转语句，条件跳转语句，标号，地址，load/store，二元操作，一元操作，强制转换和标号等组成。

REIL 是 BinNavi^[80] 逆向分析工具的中间语言，专门用来表示反汇编代码。REIL 中间语言由算术操作，位操作，数据传送操作，条件判断操作等组成。

表 3-1 是关于 RTL，LLVM IR，BIL 和 REIL 等中间代码的对比。

表 3-1 中间语言对比

中间语言 特性	RTL	LLVM IR	BIL	REIL
是否是 SSA	否	是	否	否
指令数目	55	31	17	17
扩展系数*	4	1~5	3+	20~50
应用领域	编译与反编译	编译与反编译	反编译，代码分析	逆向工程

*扩展系数指平均一条汇编指令对应中间语言指令的条数。

由汇编指令翻译成中间代码的核心是，首先构建一个<指令，中间代码列表>的词典。然后，每次解码到新指令时，到词典中查找相关指令，并将其中间代码列表取出，替换其中的参数。

但是，对于像 LLVM IR 这样的 SSA 特性的中间表达语言来说，需要在中间代码生

成时保证其 SSA 特性。具体内容可以参见 SSA 算法^[81-83]和 PTX 中间代码到 LLVM IR 的生成技术细节^[84]。

3.2 Switch 结构的优化技术

在 2.2.1 节已经介绍过编译器对 switch 结构的优化原则。本节将对在机器代码中所隐藏的两种不同的 switch 结构展开进一步分析，并设计相应的结构优化算法来重构和规整 switch 控制流子图。

3.2.1 基于跳转表的 switch 结构的优化

IDA 在分析带有跳转表的 ppc 程序时，没有有效地构建出其控制流图，从 Switch Block 基本块，并没有到各个 case 基本块的有向边。因此，在具体分析到 Switch Block 中的间接跳转指令（如 x86 的 jmp 指令，ppc 的 bctr 指令）时，需要向前分析，找到跳转表的地址 baseAddr，在 baseAddr 处找到相应的跳转表 Switch Table。跳转表中的每 4 个字节组成一个字，该字构成了相对于 baseAddr 的偏移，将该字与 baseAddr 相加得到最终的跳转地址。其计算方法可以用公式（3-1）来实现。

$$\text{realAddr} = \text{baseAddr} + \text{m}[\text{baseAddr} + 4 * \text{idx}] \quad (3-1)$$

公式（3-1）中，idx 表示跳转表的索引（有间接跳转表时为间接索引；否则为直接索引），也是 switch 的分支值，idx = 0 代表 default 分支。m[Addr]表示从内存地址 Addr 处取值；realAddr 是最终的跳转地址。

在图 3-2 的左边，从 Switch Block 基本块出发，有跳往首地址分别为 Case1_Code, Case2_Code, Case3_Code, Case4_Code 和 Case5_Code 的基本块的边，在解码阶段需要通过相应的算法，恢复出完整的 switch 控制流图，如图 3-2 的右边所示。

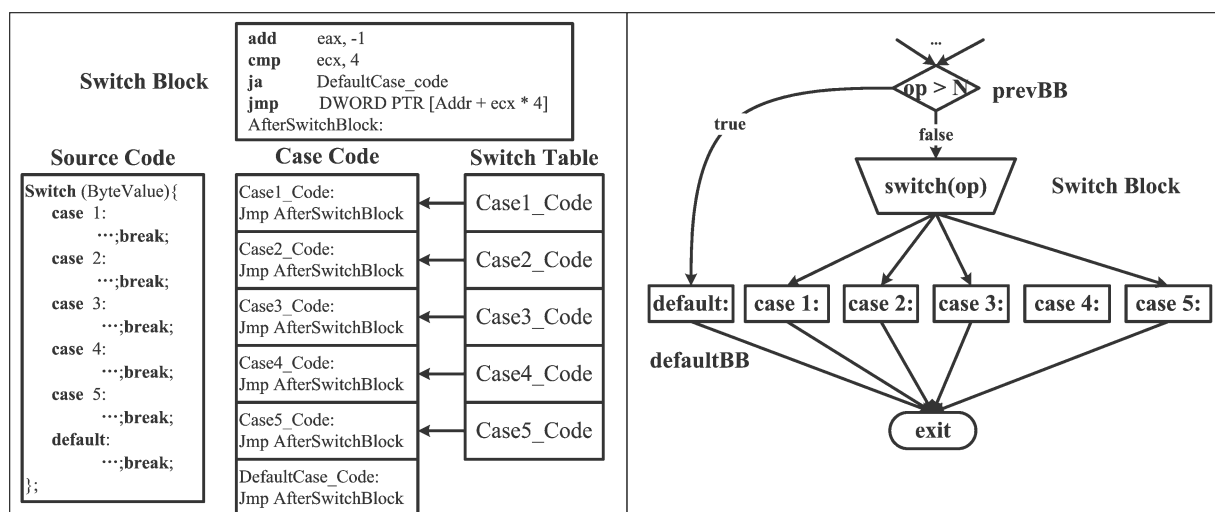


图 3-2 基于跳转表的 switch 结构翻译

在反编译基于跳转表策略生成的 switch 语句时，关键是根据间接跳转指令，直接跳转表和间接跳转表，在程序的控制流图中构造出 N-way 的分支结构。基于控制流图的中间表达式再经过控制流分析之后会自动地将 N-way 的分支结构构造成 switch 结

构, 最终经过代码生成得到 switch 语句。算法 3-2 展示了在解码阶段恢复和规整 N-way 控制流图的具体流程, 恢复算法的时间复杂度是 $O(N)$, 其中 N 为跳转表的大小; 规整算法的时间复杂度为 $O(N)$, 其中 N 为控制流图中基本块的个数。空间复杂度均为 $O(1)$ 。

Proc nway_Construct() Input: address of indirect jump, Addr Output: CFG with N-way control flow graph constructed 1. Construct N-way statement nwStmnt at Addr 2. Append nwStmnt to current basic block cb 3. Get jump table jmpTbl, its address jmpAddr 4. for each offset in jmpTbl 5. rAddr = jmpAddr + offset 6. New a basic block nb at rAddr 7. Insert nb into the CFG 8. Insert edge <cb, nb> to the CFG 9. end for end Proc	Proc nway_Format() Input: CFG Output: CFG with N-way formatted 1. for each bb in CFG 2. if bb is N-way 3. nwBB = bb 4. pBB = nwBB's predecessor 5. dBB = prevBB's successor 6. Delete edges<pBB, pBB's successors> 7. Insert Edge <nwBB, dBB> 8. Insert edges <pBB's predecessors, nwBB> 9. Delete basic block pBB 10. end if 11. end for end Proc
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

算法 3-2 基于跳转表的 switch 结构的构建和规整算法

N-way 结构的构建主要集中在解码阶段, 解码器根据被解码指令, 指令的操作码以及指令所在的上下文, 适当地构造出 N-way 结构的控制流图以及 switch 语句的分支变量。算法 3-2 中的 nway_Construct() 算法给出了如何根据 switch 指令模式和跳转表, 完成 N-way 控制流图的构建, 其中的关键部分是跳转表的查找, 跳转表的构建在加载部分完成, 以便解码阶段能够方便的得到 N-way 分支结构中的各个跳转地址, 构建完整的 N-way 控制流图。

算法 3-2 中的 nway_Construct() 算法列出了 N-way 控制流图构建算法, 即 switch 相关指令的解码以及对应控制流图的构建过程。但是, 通过算法 3-2 构建起来的 N-way 控制流图并不是完整的 switch 语句的控制流图。如图 3-3 的左图, 根据算法 3-2 中的 nway_Construct() 算法, 只构造出了由 {nwBB, dBB, case 1, ..., case N, exit} 等基本块组成的控制流图。但是, 完整的 switch 语句的控制流图应由 {pBB, nwBB, dBB, case 1, ..., case N, exit} 等基本块组成的控制流图。图 3-3 左边的控制流图, 从执行角度来看可以提高执行速度; 但是, 当被转换成高级代码时, 从阅读角度来说, 不易于理解。因此, 必须将其规整, 规整算法如算法 3-2 中的 nway_Format() 算法所示。

图 3-3 左边的控制流图是从汇编程序中构建起的完整的 N-way 控制流图, 基本块 dBB 有一条入边 <pBB, dBB>。边 <pBB, dBB> 是编译器为优化 switch 语句的执行效率和执行速度而引进的, 对于反编译没有太大的价值。如果在控制流分析之前不将边 <pBB, dBB> 和基本块 pBB 删除, 那么生成的代码中, 往往在 switch 结构外嵌套 if 结构, 影响代码理解; 因此, 将 N-way 控制流图尽早地规整化成图 3-3 右边的模式, 对于后续的控制流分析和生成代码的可读性都非常重要。N-way 控制流图规整的核心是对图 3-3

左图中控制流图的数据结构的修改，其过程如算法 3-2 中的 `nway_Format()` 算法所示。

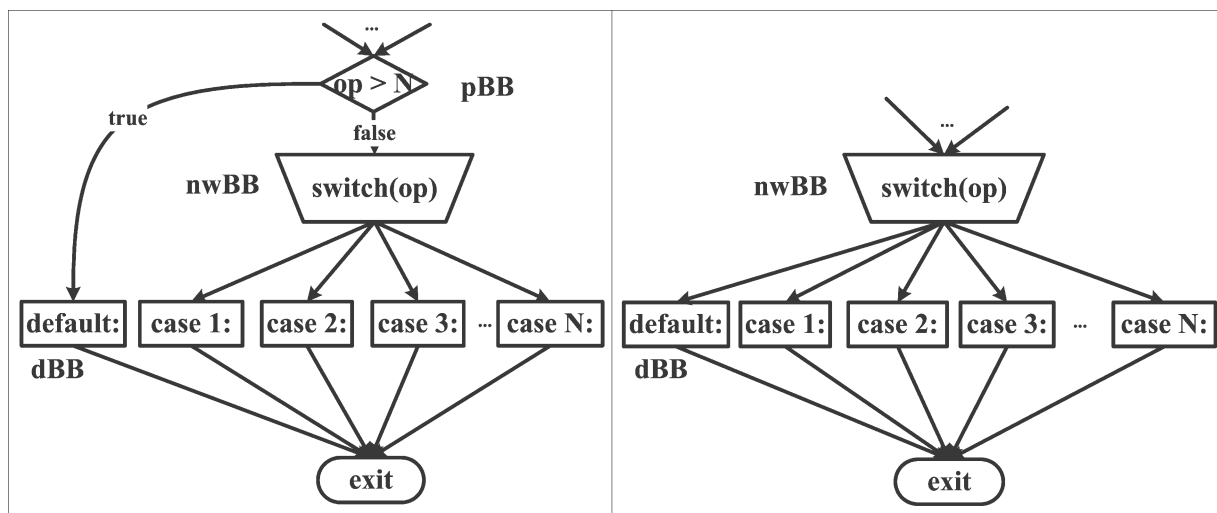


图 3-3 switch 控制结构规整示意

经过 N-way 控制流图的规整算法处理之后，最终生成的 switch 语句的可读性大大增强，去除了嵌套在 switch 结构外层的 if 结构，同时也消除了从 switch 的 default 分支跳往 if-else 处的 goto 语句。在真实的程序中，当有多个 case 分支对应于同一个基本块时，在生成 case i2, case i3, ..., case in 的代码时，通常会引入一个 goto 语句，该语句的跳转目标是 case i1 的起始位置。这样的结构也影响正常的阅读逻辑，必须在反编译过程中进行解决。最简单的解决方案是在代码生成的时候，先统计出每个 case 分支中基本块对应的 case 变量的数目；然后，输出各个 case 变量；最后，输出对应的基本块代码。通过上述方法，在实际中可以大大减少 switch 语句中的 goto 语句的数目，优化代码的结构，提高可读性。

3.2.2 基于二叉树的 switch 结构的优化

基于二叉树的 switch 结构的翻译是编译器中经常采用的技术，其目的是在 case 值分布比较离散时，根据 case 值构建二叉树，提高 switch 结构的运行速度。在分析 Hex-Rays, Phoenix, RD, REC 等多款反编译器的反编译结果时，发现它们在这方面做得很少，Hex-Rays 虽然做了一些，如图 2-2，但是其分析还不是非常完整。另外 RD 也做了一些工作，但是组合的范围不是非常大，它只是将一系列连续的判断等于常量的条件表达式组合成一个分支结构。在其余的反编译器中没有发现相应的处理。本节将给出一般的处理算法，用来规整基于二叉树的 switch 结构的优化，其具体处理过程如算法 3-3。在介绍具体算法之前，先对基于二叉树的 switch 结构的控制流图中的基本块进行分类，具体分为以下四类，循环递归定义如下：

分流基本块 (non-terminal)：出度为二，两个孩子或为终结基本块，或为分流基本块，或为普通基本块。

终结基本块 (terminal)：出度为二，且至少有一个孩子是普通基本块或多分支头基本块。

多分支头基本块 (nway-head): 二叉树结构的根节点。

普通基本块 (ordinary): 控制流图中除分流基本块, 终结基本块和多分支头基本块之外剩余的基本块。

上述的定义都是抽象定义, 在实际的检测过程中, 可以基于基本块自身的特征去确定每个基本块的初始类型, 然后再根据其所处上下文, 去判断该基本块的真正类型。将由分流基本块 (non-terminal) 和终结基本块 (terminal) 所组成的集合叫分支集合, 将普通基本块的集合称为 case 集合。

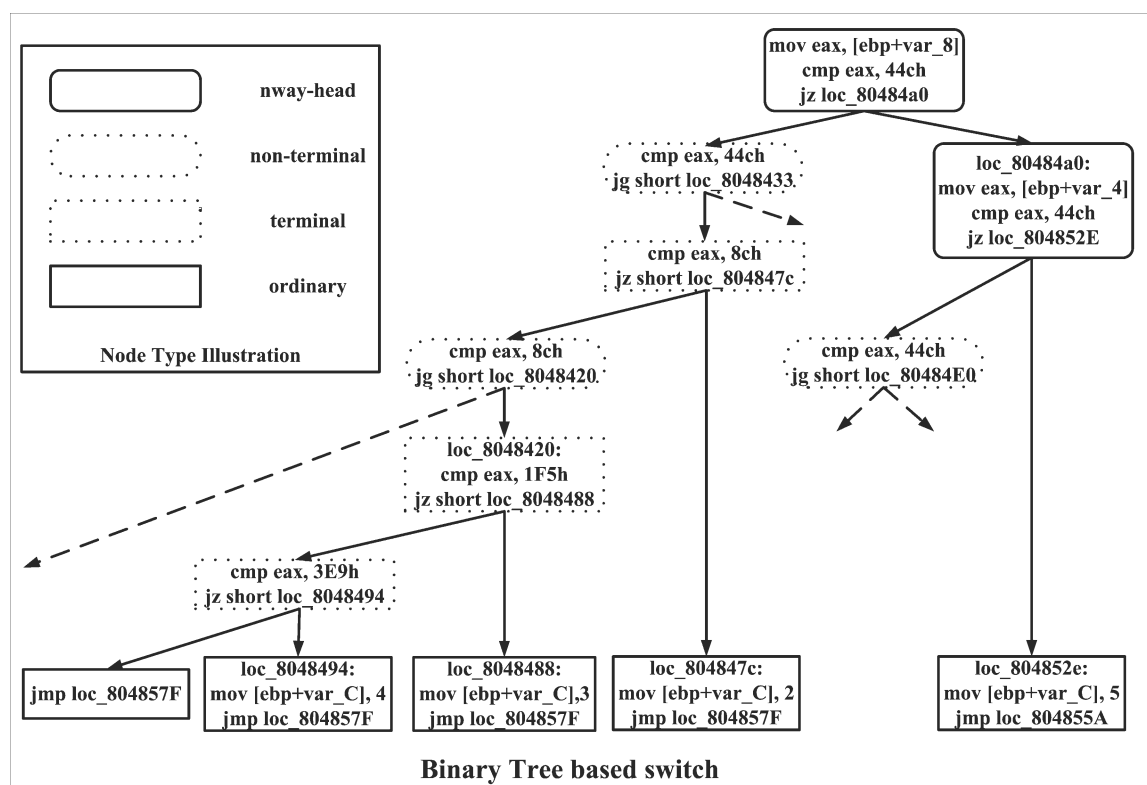


图 3-4 基于二叉树的 switch 结构中的结点分类

图 3-4, 展示了真实程序的一部分控制流子图, 其中包含了一个基于二叉树的 switch 结构的部分流图。从图中可以看出, 多分支头基本块 (nway-head) 是基于二叉树的 switch 结构中的第一个基本块, 其功能是将分支变量赋值给 `eax` 寄存器, 并将 `eax` 与某一常量进行比较, 用比较结果确定分支走向。图 3-4 中存在两个多分支头基本块, 并且二者在结构上是嵌套关系。分流基本块 (non-terminal) 是决策结点, 它不跳往任何具体代码块, 只是辅助二叉树的查找。终结基本块 (terminal) 是匹配结点, 决策到了该分支结点之后, 已经可以对应一个具体的控制流子图, 用于完成一个 case 分支的功能。普通基本块 (ordinary) 是一个具体的代码片段, 也是 switch 结构中 case 分支所对应的控制流子图的入口点。算法 3-3 遍历程序控制流图中的每一个基本块, 因此其时间复杂度是 $O(N)$, 其中 N 为控制流图中基本块的数目。

Proc binary_tree_based_switch_recovery()

/*Parameter and Return Illustration*/

Input: a procedure proc expressed in CFG form
Output: a procedure with binary tree based switch recovered

```

0.  /*Step 1*/
1.  Sort the basic blocks in proc in ascending order according to their ADDRESS
2.  /*Step 2*/
3.  for each basic block B of proc
4.      if B is a branch
5.          branch_list = NULL    // store all the non-terminal and terminal basic blocks
6.          case_map = NULL      // store all the <case_value, basic block > pairs
7.          unvisited_list = NULL // facilitate the binary tree traversal process
8.          /*Step 2.1 find all the binary tree based switch*/
9.          Initiate unvisited_list with B
10.         while unvisited_list is not empty
11.             Bx = unvisited_list.pop()
12.             if Bx is a terminal node
13.                 Get the case value case_value from instructions in Bx
14.                 Get the jump target By of Bx
15.                 Insert <case_value, By> into case_map
16.                 Get the child of Bx which is not By, Bz
17.                 Insert Bz into branch_list
18.                 Insert Bz into unvisited_list
19.             else if Bx is a non-terminal node
20.                 Insert both children of Bx into branch_list
21.                 Insert both children of Bx into unvisited_list
22.             end if
23.         end while
24.         /*Treat the binary tree with size smaller than 3 as an ordinary if-else structure*/
25.         if the length of case_map is less than 3
26.             continue
27.         end if
28.         /*Step 2.2 update the binary tree based switch with N-WAY structure */
29.         for each basic block Bx in branch_list
30.             Delete all the incoming and outgoing edges of Bx and Bx itself
31.         end for
32.         Convert B into a N-WAY basic block
33.         for each <case_value, Bx> in case_map
34.             if Bx is not a jump basic block
35.                 Add edge <B, Bx> with case_value as its case value
36.             else
37.                 Delete Bx and all its outgoing edges
38.             end if
39.         end for
40.     end if
41. end for
end Proc

```

算法 3-3 基于二叉树的 switch 结构规整算法

算法 3-3 的第一步是将过程体中的基本块按照地址进行排序，这样做的目的是为后续分析提供方便，因为属于同一个 switch 语句的各个判断语句所在基本块在地址上都是相连的，而且地址最低的通常是二叉树的入口处。因此，找到了多分支头基本块（nway-head），整个二叉树可以通过广度优先遍历或深度优先遍历的方式获取。算法 3-3 采用广度优先遍历的方式，查找整个 switch 结构中的分支集合和 case 集合，并将分支集合以及分支集合中的所有边集删除。对于 case 集合，首先要删除其中的间接跳转基本块，然后再将剩余的基本块与相应的多分支头基本块（nway-head）组合成多分支结构。

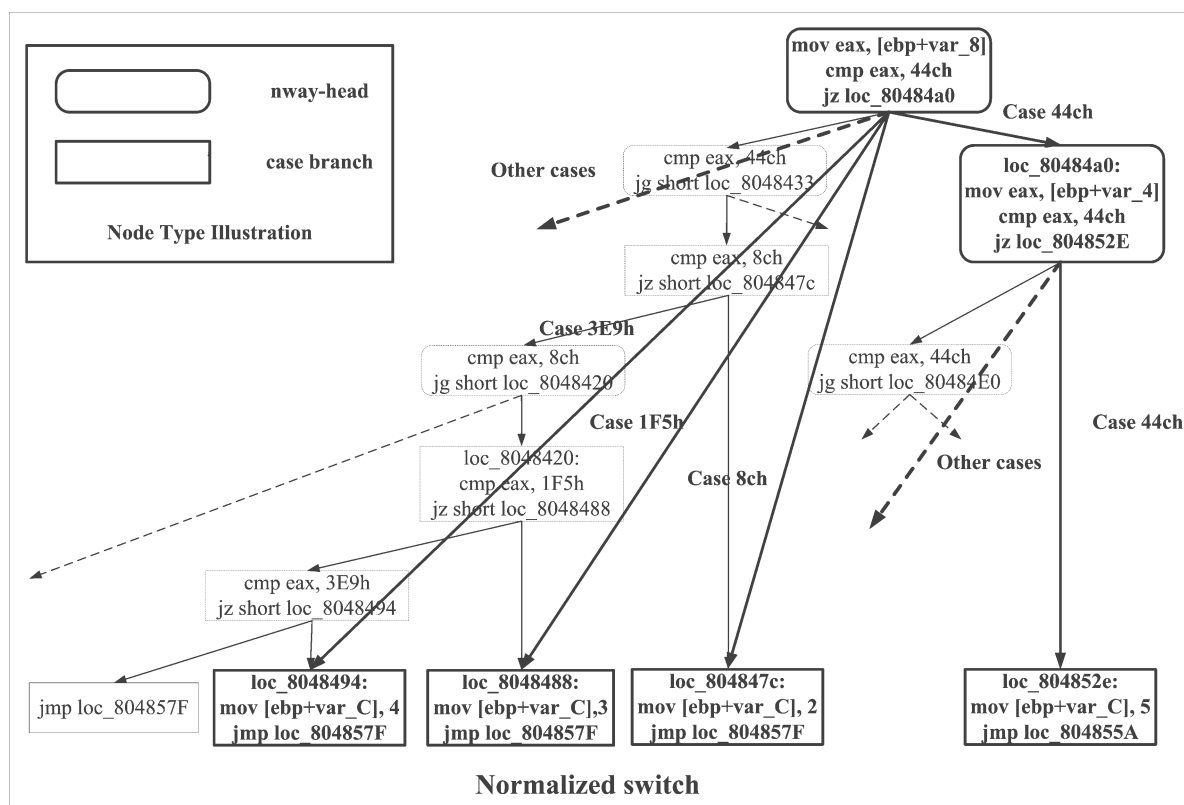


图 3-5 基于二叉树的 switch 结构规整后的结构

图 3-5 中的虚线阴影部分展示的是原始的 switch 结构中的基本块以及相应的控制边，这些基本块和边组成的控制流子图经过算法 3-3 的处理之后被删除。而黑色着重线部分是重构后的多分支结构。从图 3-5 中可以看出，此图中存在着一个嵌套的 switch 结构，在最外层的 case 44ch 分支内部又嵌套着另一个 switch 结构。从算法 3-3 的设计来看，该算法能够处理这种嵌套结构，体现出了算法 3-3 的优越性。

3.3 Goto 语句消除技术

根据^[47]实验结果，Hex-Rays 对 p2p Zeus^[85]样本进行反编译，得到 49,514 行代码，产生 1,571 个 goto 语句，平均每 32 行就有一个 goto 语句。因此，有必要对 goto 语句引入的原因以及 goto 语句的删除进行仔细研究。本节首先分析 goto 语句的引入原因，然后利用基于代码复制的方法，消除程序中的部分 goto 语句。该方法在实践中具有一

定的意义。

3.3.1 Goto 语句分析

程序中的 goto 语句使程序的逻辑混乱不清。同样，在反编译的结果中也要避免过多地生成 goto 语句。在分析了大量产生 goto 语句的过程体的中间表达式后，发现 goto 语句的引入可以从控制流图上找到对应的结构。

第 2.3.1 节已经介绍了编译领域中，优化程序中的 goto 结构的方法。在反编译领域中，goto 语句的优化和消除同样非常重要，毕竟反编译的结果是面向人，而人对于非结构化的程序的理解能力往往不是很好。

通过总结分析，在控制流图上表现出的 goto 结构往往是由于编译器在优化复合条件表达式，循环，break 语句，continue 语句和 goto 语句等引入。随后，由于结构化算法分析不是非常精确，导致在反编译结果中出现 goto 语句。

Hex-Rays 反编译器的 goto 语句消除集中在伪代码变换阶段。在该阶段，通过创造 for 循环，break/continue 等语句，逐步消除更多的 goto 语句。可见，Hex-Rays 的 goto 消除主要通过增加 break/continue 等语句，消除多出口循环和多入口循环等不规则结构。

Boomerang 反编译器的 goto 语句消除，主要通过表达式组合和结构化算法来实现 goto 语句的消除。并且在代码生成时，只要遇到不可归约控制流图，代码生成器就会生成一个 goto 语句，使得代码生成得以继续。JAVA 反编译器 Krakatoa^[86]，利用 Ramshaw 的 goto 消除技术重构程序控制流图，达到 goto 语句消除的目的。

第 3.3.2 节介绍的代码复制消除 goto 语句方法虽然不是结构化算法中的一部分，但是它可以提高结构化算法的分析结果，提高程序的结构抽象程度。而且代码复制消除 goto 语句是许多后续研究的基础，尤其是在 Phoenix 反编译器和 DREAM 反编译器中，代码复制被广泛使用。

3.3.2 代码复制消除 goto 语句

图 3-6 展示了有可能产生 goto 结构的控制流子图模式，消除 goto 结构的方法以及由简单复制引起的级联复制。

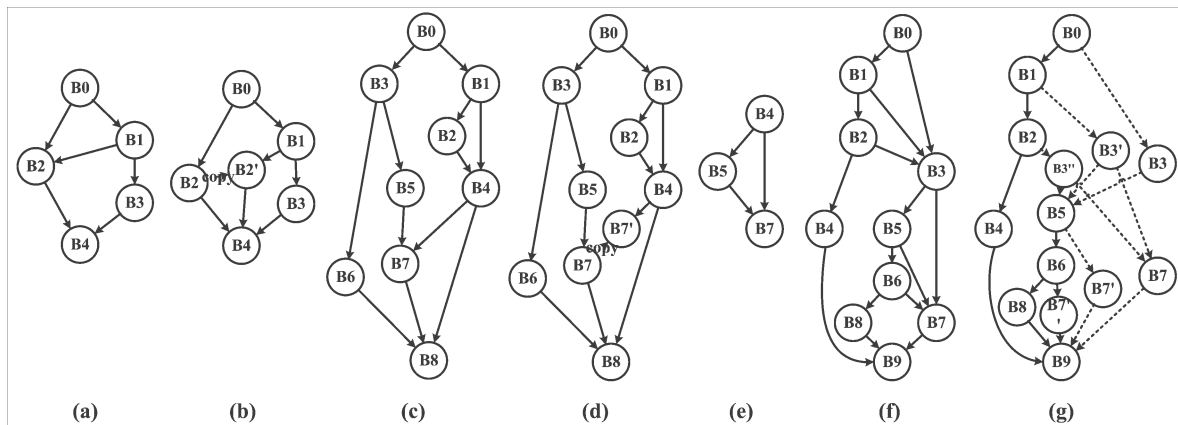


图 3-6 代码复制消除 goto 语句示意

在图 3-6 中, 图 3-6(a)是实际中遇到的一个例子, 具有这种结构的程序, 如果不进行表达式组合, 在生成高级代码时会产生 goto 语句。理想情况下, 不同分支的基本块之间不应当有边相关联, 即在控制流图中不应当存在有向边 $\langle B1, B2 \rangle$ 。图 3-6(a)的控制流图是因在编译时期, 编译器对于组合条件表达式采用短路算法而引入的, 文献^[3]给出了条件表达式组合的子图模式, 但是归结起来可以用图 3-6(a)概括。如果 B0 和 B1 的条件表达式是与逻辑, 则 B2 是 false 跳转基本块; 如果 B0 和 B1 的条件表达式是或逻辑, 则 B2 是 true 跳转基本块。

如果通过代码复制, 仅消除图 3-6(a)引入的 goto 结构, 那么采用文献^[3]中提到的子图模式检测算法已经可以满足需求。但是, 有时候程序的控制流图比较复杂, 如图 3-6(c)中所示的结构, 有向边 $\langle B4, B7 \rangle$ 并非是组合条件表达式采用短路算法而引入的结构。但是, 同样由于该有向边的存在, 会在生成代码中引入 goto 语句。因此, 必须在代码生成之前将该有向边消除。观察图 3-6(a)和图 3-6(c), 可以总结出一些特征, 利用这些特征在控制流图中检测 goto 结构, 并通过代码复制的方法消除 goto 结构, 如图 3-6(b)和图 3-6(d)所示的方法。以图 3-6(c)为例, 得到如下四条启发式规则:

- (1) 基本块 B4 是分支节点, 即 $\text{outDegree}(B4)=2$
- (2) 基本块 B7 是汇合且非分支节点, 即 $\text{inDegree}(B7) \geq 2 \ \&\& \ \text{outDegree}(B7)=1$
- (3) 基本块 B7 是基本块 B4 的后继节点, 即 $B7 \in \text{succ}(B4)$
- (4) 基本块 B4 不是基本块 B7 的前向支配结点, 且 B7 不是 B4 的后向支配结点

启发式规则 (1) 和 (2) 是对图 3-6(c)中的基本块 B4 和 B7 的度的数值关系的描述; 规则 (2) 中的条件限制 $\text{outDegree}(B7)=1$ 非常重要, 如果不做这一限制, 在某些情况下, 将会引入更多的 goto 结构, 如图 3-6(f)和图 3-6(g)所示。

对于图 3-6(f)中的控制流图, 如果不限限制被复制基本块的出度, 则会复杂化程序的控制结构, 增加控制流分析复杂度, 影响生成代码的可读性。图 3-6(f)中有四条交叉边, 分别是 $\langle B1, B3 \rangle$, $\langle B2, B3 \rangle$, $\langle B5, B7 \rangle$ 和 $\langle B6, B7 \rangle$ 。图 3-6(g)是未对被复制基本块的类型进行限制而采用代码复制得到的程序控制流图, 在原有的四条交叉边被消除的同时, 又引入了新的交叉边。例如, 有向边 $\langle B3, B5 \rangle$, $\langle B'_3, B_5 \rangle$ 和 $\langle B''_3, B_5 \rangle$ 等边又成为新的交叉边, 这些边在随后的迭代过程中要继续被消除, 引发大量的复制。这样的级联复制会使程序的控制流图比较复杂, 生成的代码的冗余度大大增加, 并有可能引入更多的 goto 语句, 进而影响最终生成的程序的可读性。因此, 必须对被复制基本块的类型进行限制。

规则 (3) 描述了存在一条有向边 $\langle B4, B7 \rangle$; 规则 (4) 则排除如图 3-6(e)中的子图模式, 图 3-6(e)中的有向边 $\langle B4, B7 \rangle$ 满足启发式规则 (1), (2) 和 (3), 但是该有向边并不是交叉边。

整个 goto 结构检测算法遵从规则 (1) 到规则 (4) 的检验顺序, 即整个判断逻辑是与的关系。逻辑判断式为

$$\text{isGoto} = \text{pCond}(1) \wedge \text{pCond}(2) \wedge \text{pCond}(3) \wedge \text{pCond}(4)$$

$pCond(i)$ 表示第 i 条规则。如果 $isGoto=true$ ，则边 $\langle B_4, B_7 \rangle$ 是交叉边，否则是正常的边。算法的时间复杂度为 $O(N)$ ，其中 N 为控制流图中的基本块个数。空间复杂度为 $O(1)$ 。通过迭代地使用上述四点启发式规则，可以很好的在控制流图中检测出 `goto` 结构。当 `goto` 结构被检测出之后，按照图 3-6(d)将基本块 B_7 复制一份，得到孤立的基本块 B'_7 ；接着，修改有向边 $\langle B_4, B_7 \rangle$ 为有向边 $\langle B_4, B'_7 \rangle$ ；最后，将 B_7 的出边赋给 B'_7 ，作为 B'_7 的出边。

代码复制的时间非常关键，不能够在解码结束后立刻进行代码复制，过早的复制会导致被复制的基本块中的被定值变量在数据流分析时处于不同的上下文中，从而在代码生成时，导致代码不一致，影响代码理解。例如，假设在图 3-6(c)的基本块 B_7 中定义了变量 x ，如果在解码结束，立即进行代码复制，那么变量 x 就存在两份，记为 x_{B_7} 和 $x_{B'_7}$ 。那么在进行数据流分析中， x_{B_7} 所处的上下文是 $\langle B_0, B_3, B_5, x_{B_7}, B_8 \rangle$ ，而 $x_{B'_7}$ 所处的上下文是 $\langle B_0, B_1, B_2, B_4, x_{B'_7}, B_8 \rangle$ 。最终，经过复制传播和表达式传播等数据流优化算法，可能导致 $x_{B_7} \neq x_{B'_7}$ 。代码复制的最佳时间是在控制流分析之前，这样既能保证代码的一致性，同时还能保证程序的结构清晰。图 3-7 是经过代码复制后的结果与 Hex-Rays, DREAM 反编译器的结果的对比示意。

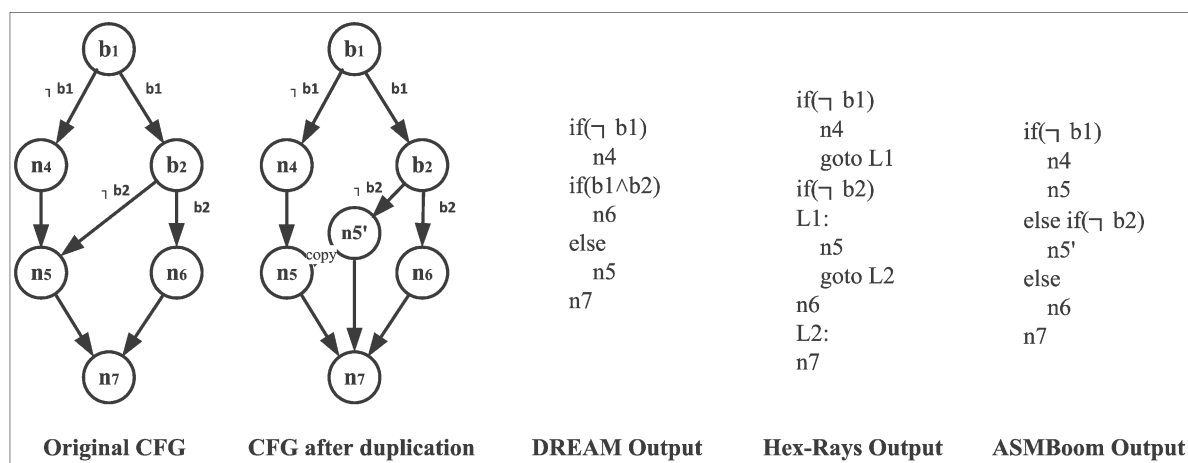


图 3-7 代码复制消除 `goto` 语句以及结果对比示意

图 3-7 中的原始控制流图中的有向边 $\langle b2, n5 \rangle$ 满足 `goto` 结构的四个条件。因此，按照算法的步骤，基本块 $n5$ 将被复制，得到新的控制流图。DREAM Output 是 DREAM 反编译器对 Original CFG 的反编译结果；Hex-Rays Output 是 Hex-Rays 反编译器对 Original CFG 的反编译结果；ASMBoom Output 是经过代码复制，由 ASMBoom 反编译器产生的反编译结果。对比三个反编译器的输出结果发现，DREAM 反编译器的输出结果最好，尤其是在复合条件表达式组合方面做得比较好；Hex-Rays 在复合条件表达式组合方面不是非常理想，而且引入了两个 `goto` 语句，程序结构比较差。ASMBoom 反编译器的输出结果中复合条件表达式被嵌套的 `if-else` 结构取代，代码中存在着部分冗余，但其结构的可读性要比 Hex-Rays 反编译器的结果好。

总之，通过代码复制可以消除部分 goto 语句，增加代码的可读性。但是，这种策略适合于复制顺序基本块，而不适合复制分支基本块；如果待复制的基本块是分支基本块，则最好采用条件表达式组合的方式优化控制流图的结构。另外，代码复制的时间应当在控制流分析之前进行，过早的复制会导致代码的不一致性。

3.4 固有函数的消除技术

本节从固有函数的分析和固有函数的消除算法入手，突出子图同构算法在相关领域的应用和价值。

3.4.1 固有函数分析

固有函数是编译器的内建函数，独立于具体的库和系统，是编译器所特有的。固有函数通常被内联插入到代码中以避免函数调用的开销，从而用高效的代码段代替函数调用。由于固有函数是编译器所特有的，因此编译器非常清楚固有函数的性能，而且可以根据固有函数所处的上下文以及固有函数的参数，对于固有函数的展开以及内存分配区别对待。固有函数可以辅助编译器做类型检查，寄存器分配，指令调度和函数调用栈的维护等。

有些固有函数是和处理器密切相关的，这些固有函数的可移植性较差。而还有一些固有函数是独立于处理器的，这些函数的可移植性相对好一些。

固有函数的代码以及固有函数的优化特征，可以反映每一类编译器的特性。因此，做好固有函数的检测，可以作为编译器识别的重要依据。表 3-2 列出了 Visual Studio C++编译器，GCC 编译器，Intel C++编译器以及 LLVM 编译器在固有函数处理方面的特点。

表 3-2 主流编译器中固有函数对比分析

编译器 特性	VC++	GCC	Intel C++	LLVM
体系结构种类	arm, x86, x64	x84, ppc, arm, mips, Alpha 等	IA-32, Intel 64, IA-64	--
固有函数种类	CRT(C Run-time Library)函数以及体系结构相关的一些操作。	向量指令，带溢出检测的算术运算，对象类型检测，指针检测，标准 C 库以及与体系结构相关的一些操作。	整数算术运算，浮点运算，字符串和块拷贝，SSE4，SSSE3，SSE3，SSE2，SSE，数据对齐，内存申请等机器相关操作。	参数处理，垃圾回收，代码生成，标准 C 库，位操作，算术运算，浮点运算，调试，异常处理等。

从表 3-2 分析，可以看出固有函数遍布在编译，调试，机器管理以及具体运算的各个方面，如果可以有效地规约固有函数，那么对于反编译结果可读性的提高将非常有益。有许多反编译器在内联固有函数的识别方面已经做了一些工作。Hex-Rays 反编译器和 RD 反编译器在内联固有函数的识别方面做了一些工作。Hex-Rays 的 F.L.I.R.T

技术^[87]和 RD 的技术^[24]都是从指令流中提取相关固有函数的特征，而忽略指令流的结构特征。在实际识别中，效果不是非常理想。

图 3-8 的左上部分是源码，左下部分是 Hex-Rays 反编译器的输出的部分结果，右半部分是源码中内联固有函数 `memcmp` 的部分汇编代码片段及其控制流子图，可以看出其控制流图非常复杂。

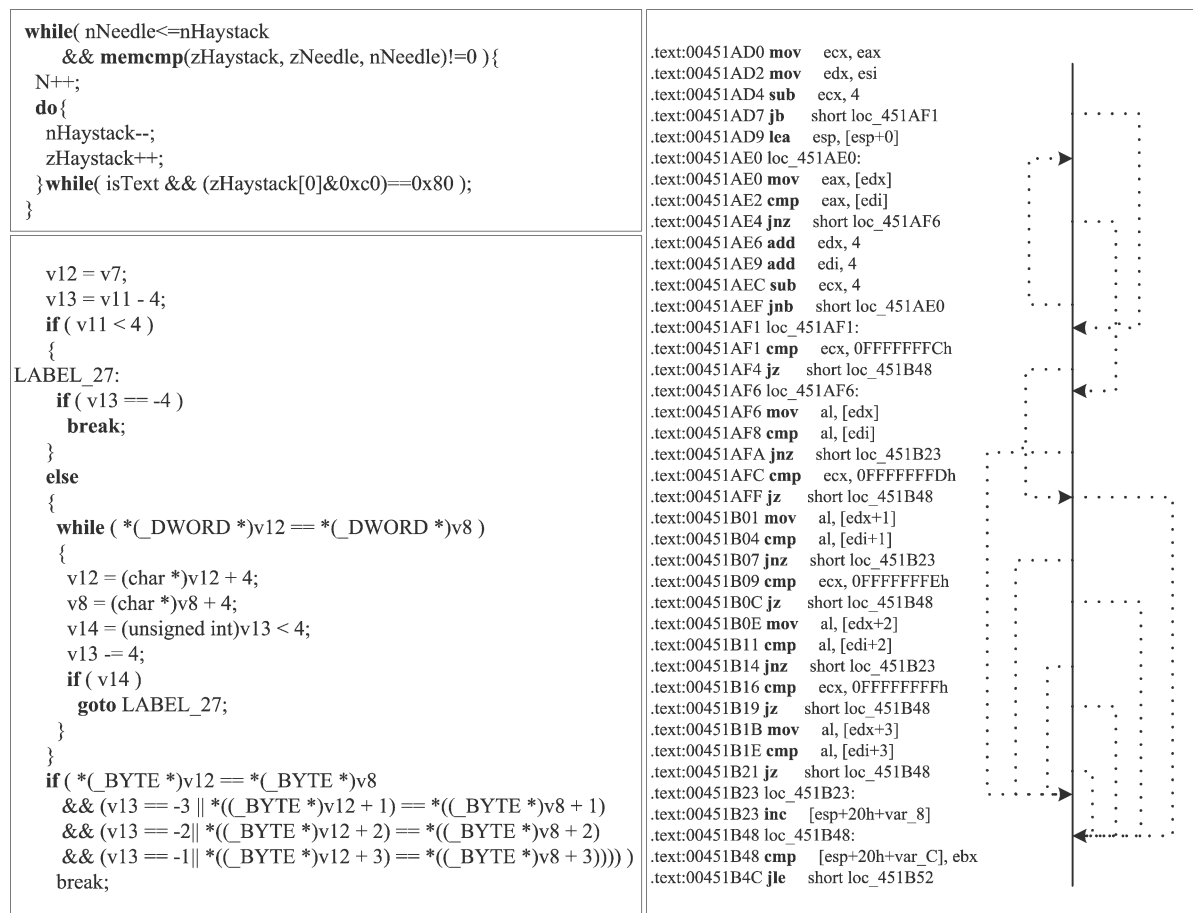


图 3-8 Hex-Rays 没有规约的内联固有函数

Hex-Rays 的反编译结果中包含有 `goto` 语句和复杂的条件判断语句，而这些复杂的代码段其实可以抽象成一个函数调用语句 `v13 = memcmp(v12, v8, v11)`。但是，Hex-Rays 没有将该语句分析出来，可见 Hex-Rays 所采用的 F.L.I.R.T 技术在某种程度上存在缺陷，这也是我们提出基于子图同构的固有函数消除算法的原因。

3.4.2 基于子图同构的内联固有函数消除算法

算法大致包括三部分，分别是基本块的语义匹配，匹配过程以及参数和返回值的确定。基本块的语义匹配用来判断两个基本块的语义是否相似。因为，在许多情况下，尽管两个控制流图在结构上是同构的，但是其程序语义确实完全不同的，其根本原因就在于基本块的语义存在差异。匹配过程用来确定目标图与哪个模板图是相匹配的。当模板图在目标图上被匹配到之后，需要确定被内联函数的参数及其返回值。下图列出了整个算法的流程。

Proc intrinsic_Function_Reduction()**INPUT:** target function in CFG format; list of template template_list**OUTPUT:** target function with intrinsic function reduced to a single call

```

1. for each template  $\in$  template_list
2.     Construct match state  $S$  for target and template , set mapping  $M(S) = \emptyset$ .
3.     Initiate the state stack state_stack with  $S$ .
4.     while state_stack is not empty
5.          $S = \text{Pop}(\text{state\_stack})$ .
6.         if  $M(S)$  covers all the nodes then
7.             Determine the parameter and return value of the inlined function.
8.              $\text{Pop}(\text{state\_stack})$ 
9.         else
10.            Compute the set  $P(S)$  of the pairs candidate for inclusion in  $M(S)$ .
11.            for each  $p \in P(S)$ 
12.                if both the feasibility rules and semantic compatibility succeed for
13.                    the inclusion of  $p$  in  $M(S)$ 
14.                then
15.                    Compute the state  $S'$  obtained by adding  $p$  to  $M(S)$ .
16.                    if  $M(S')$  covers all the nodes then
17.                        Determine the parameter and return value of the inlined function.
18.                    else
19.                         $\text{Push}(\text{state\_stack}, S')$ .
20.                    end if
21.                end if
22.            end for
23.        end if
24.    end while
25. end for
end Proc

```

算法 3-4 内联固有函数消除算法

算法 3-4 中列出了内联固有函数的消除算法，原始的递归式的 VF 匹配算法被转换成了迭代式的算法。它的时间复杂度与 VF 算法是一样的，但是它的空间复杂度要远远低于 VF 算法，因为运行时的栈开销被大大减小。程序中的完全映射 M 是一系列有序对的集合，每一个有序对代表图 $G_1 = (N_1, B_1)$ 中的结点 n 与图 $G_2 = (N_2, B_2)$ 中的结点 m 的匹配，其形式化表示如下

$$M = \{(n, m) \in N_1 \times N_2 \mid n \text{ is mapped onto } m\} \quad (3-2)$$

状态空间表示 (SSR) 被用来描述图的匹配过程，每一个状态代表匹配过程中的部分解决方案。部分解 $M(S)$ 是在状态 S 处的解，它只是完全映射 M 的一个子集。

基本块的语义匹配

有许多用来判断基本块语义匹配的方法。这是一个在编译优化和恶意代码分析领域，广泛被研究的话题。文献^[49]利用基本块的指令混淆比向量之间的 Manhattan 距离来计算两个基本块之间的相似度。文献^[88]通过两个基本块的指令序列流之间的编辑距

离来计算它们的相似度。文献^[89]通过基本块的符号执行结果来判断它们的相似性。文献^[90]根据基本块中指令的类别给基本块指定一个颜色。他们将指令分为七类，分别是数据传输指令，算术指令，逻辑指令，测试指令，栈指令，分支指令和函数调用指令。最终，基本块语义相似问题被转换成基本块的颜色相似性判断问题。

但是，上面所有的方法都是基于统计的。在 ASMBoom 系统中，由于基本块的语义不仅仅依赖于其指令序列的统计特性，还依赖其指令的依赖关系。因此，我们采用子序列的方式来原因基本块的语义相似性。例如集合 $A=\{a_1, a_2, \dots, a_n\}$ ，集合 $B=\{b_1, b_2, \dots, b_m\}$ ，集合 A 语义相似于集合 B，如果存在 B 的一个有序子序列 $B_1=\{b_{i1}, b_{i2}, \dots, b_{in}\}$ ， $|A| = |B_1|$ 且 $\forall k \in \{1, 2, \dots, n\}$ ，有 $a_k = b_{ik}$ 。采用子序列的好处，一则可以保证指令的顺序和指令操作码相匹配，进而保证数据流是相匹配的；二则可以处理边界情形，因为目标子图的边界通常和其它非目标子图的基本块中的指令混杂在一起。

匹配过程

匹配过程是使用 VF 子图同构算法，确定模板在目标函数出现位置的过程。图 3-9 展示了整个动态匹配过程。在例子中，目标过程体 G1 由四个基本块构成，G2 由三个基本块构成。目标过程体是一个简单地计算两个整数的最大值的函数，而模板是 max 宏的展开。从搜索树来看，由于采用了启发式的搜索策略，许多搜索分支被剪枝。

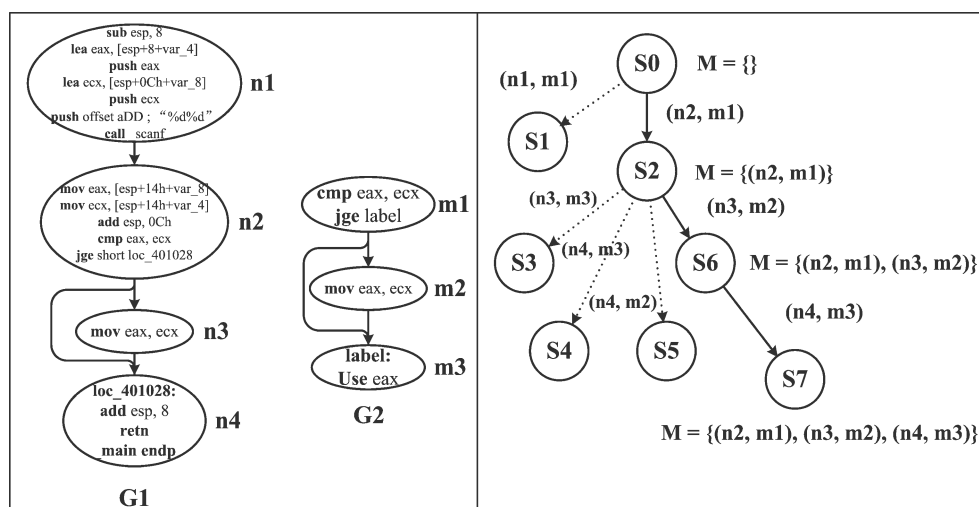


图 3-9 子图匹配过程示意

图 3-9 展示了目标图 G_1 和模板图 G_2 的匹配过程。初始时，匹配解 M 为空，状态空间 SSR 为 S_0 。首先，检测 n_1 与 m_1 是否匹配，SSR 由 S_0 变为 S_1 。由于它们的语义不匹配，所以匹配失败，SSR 由 S_1 返回 S_0 。接着 n_2 与 m_1 匹配，SSR 由 S_0 变为 S_2 ，由于 n_2 和 m_1 的语义相匹配，因此匹配继续，匹配解 M 变为 $\{(n_2, m_1)\}$ 。在匹配状态 S_2 ， n_2 的每一个后继结点与 m_1 的每一个后继结点进行匹配。当 SSR 试图从 S_2 扩展到 S_3 时，虽然 n_3 和 m_3 的度数相等，但是由于 n_3 和 m_3 语义不匹配，所以匹配状态由 S_3 返回到 S_2 。接着 SSR 从 S_2 扩展到 S_4 ，由于 n_4 和 m_3 的度数相等，且 n_4 和 m_3 的语义相匹配，因此匹配解 M 变为 $\{(n_2, m_1), (n_4, m_3)\}$ 。当 SSR 从 S_4 继续扩展时，由

于 n_4 和 m_3 都没有后继，且匹配解 M 既没有覆盖 G_1 也没有覆盖 G_2 。因此，匹配失败，SSR 从 S_4 返回到 S_2 。当 SSR 从 S_2 扩展到 S_5 时，由于 n_4 与 m_2 的语义不匹配，所以扩展失败，SSR 由 S_5 返回到 S_2 。

最后， n_3 与 m_2 相匹配，二者度数相等且语义相匹配，因此，SSR 从 S_2 扩展到 S_6 ，匹配解 M 变成了 $\{(n_2, m_1), (n_3, m_2)\}$ 。在状态 S_6 ， n_3 的每一个后继结点与 m_2 的每一个后继结点相匹配。结点 n_3 有唯一的后继结点 n_4 ，结点 m_2 有唯一的后继结点 m_3 ，且由于 n_4 与 m_3 的语义相匹配，因此，SSR 由 S_6 扩展到 S_7 ，匹配解 M 变成 $\{(n_2, m_1), (n_3, m_2), (n_4, m_3)\}$ 。此时，由于 n_4 和 m_3 均无后继，且 M 已经完全覆盖了 G_2 ，因此完成一次匹配，并找到了匹配解 $M = \{(n_2, m_1), (n_3, m_2), (n_4, m_3)\}$ ，即 G_1 中的 n_2 与 G_2 中的 m_1 相匹配， G_1 中的 n_3 与 G_2 中的 m_2 相匹配， G_1 中的 n_4 与 G_2 中的 m_3 相匹配。

在匹配完成之后，下一步最重要的是，将 G_1 中由 $\{n_2, n_3, n_4\}$ 组成的子图进行抽象并确定其参数和返回值。

确定参数和返回值

在确定具体参数和返回值前，需要确定内联固有函数的边界并重构目标函数的控制流图。对于内联固有函数的入口基本块，由于入口基本块的指令经常与其前驱基本块的指令混合在一起。在这种情况下，我们需要将入口基本块进行分裂，其中一个基本块 A 不包括任何与模板中的指令相关的指令，而另一个基本块 B 只包含和模板中的指令相关的指令。同时，增加一从基本块 A 到基本块 B 的边，将原始基本块的前驱赋给基本块 A 。而基本块 B 作为内联固有函数的入口，最终将被转换成只包含一个函数调用的基本块。

对于内联固有函数的退出基本块，所有被匹配的基本块的跳转目标基本块组成了基本块 B 的后继结点。图 3-10(a)是一个嵌入了 `strcmp` 模板函数的程序的一部分，`strcmp` 模板的基本块在程序中用虚线框标注。图 3-10(b)展示了 `strcmp` 模板函数。图 3-10(c)展示了在内联固有函数 `strcmp` 被消除之后，目标函数的表示形式。

在图 3-10 中的目标函数和 `strcmp` 固有函数之间存在一个映射 $\{(B_2, T_1), (B_3, T_2), (B_4, T_3), (B_5, T_4), (B_6, T_5), (B_7, T_6)\}$ 。因此，目标函数中的基本块集合 $\{B_2, B_3, B_4, B_5, B_6, B_7\}$ 将会被消减成一个基本块 B_2' 。所有跳入由 $\{B_2, B_3, B_4, B_5, B_6, B_7\}$ 构成的子图的边集为 $\{<B_1, B_2>\}$ ，而所有跳出由 $\{B_2, B_3, B_4, B_5, B_6, B_7\}$ 构成的子图的边集为 $\{(B_6, B_8), <B_7, B_8>\}$ 。由于 $<B_6, B_8>$ 和 $<B_7, B_8>$ 的目标基本块都是 B_8 ，因此基本块 B_2' 只有一个出边。最终，目标函数被消减成只有三个基本块 B_1, B_2' 和 B_8 的函数，边集为 $<B_1, B_2'>$ 和 $<B_2', B_8>$ ，如图 3-10 中的子图 (c) 所示。

目标函数的控制流图被消减之后，剩下最重要的工作是内联固有函数的参数和返回值的识别。根据^[15]的计算方法，参数和返回值的计算如下

$$\begin{aligned} params(p) &= live_on_entry(p) \cap param_filter(p) \\ results(p) &= return_filter(p) \cap live_on_exit(p) \end{aligned} \quad (3-3)$$

被调用过程体 p 的参数 $params(p)$ 由函数调用点处的活跃变量集合 $live_on_entry(p)$

和参数过滤器 $\text{param_filter}(p)$ 的交集组成。被调用过程体 p 的返回值 $\text{results}(p)$ 由返回值过滤器 $\text{return_filter}(p)$ 和函数调用点后面的活跃变量集合 $\text{live_on_exit}(p)$ 的交集组成。

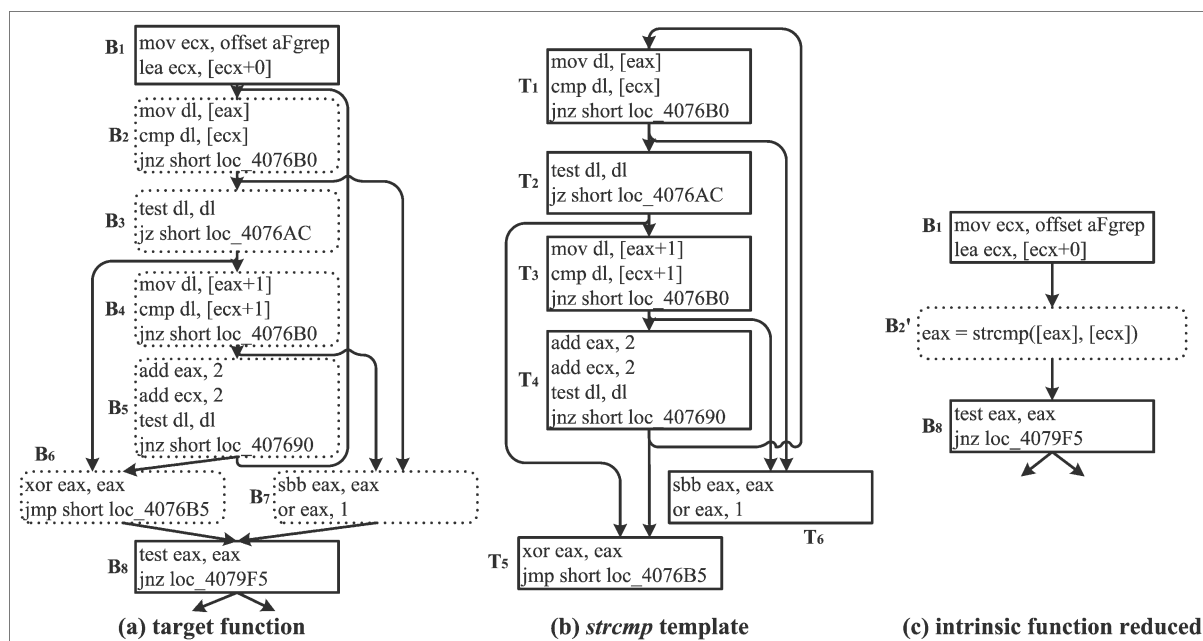


图 3-10 内联函数参数和返回值分析

$\text{Live_on_entry}(p)$ 和 $\text{live_on_exit}(p)$ 通常由活跃变量分析算法计算得到。但是，在固有函数的参数和返回值分析中，由于其参数和返回值已经被隐藏在代码中。因此，我们将 $\text{live_on_entry}(p)$ 和 $\text{live_on_exit}(p)$ 都设置为全集，即 $\text{params}(p)$ 完全由 $\text{param_filter}(p)$ 决定，而 $\text{results}(p)$ 完全由 $\text{return_filter}(p)$ 决定。

Param_filter 和 return_filter 由一系列三元组 $\langle \# \text{BasicBlock}, \# \text{Instruction}, \# \text{Operand} \rangle$ 决定，其中 $\# \text{BasicBlock}$ 表示基本块的编号， $\# \text{Instruction}$ 表示指令编号， $\# \text{Operand}$ 表示操作数编号。例如在图 3-10(b)中， strcmp 固有函数的 param_filter 是 $\{ \langle 1, 1, 2 \rangle, \langle 1, 2, 2 \rangle \}$ ，意思是 strcmp 固有函数的参数可以从第 1 个基本块中的第 1 条指令的第 2 个操作数和第 1 个基本块中的第 2 条指令的第 2 个操作数中获得。 Strcmp 固有函数的 return_filter 是 $\{ \langle 6, 2, 1 \rangle \}$ ，意思是 strcmp 固有函数的返回值可以从第 6 个基本块中的第 2 条指令的第 1 个操作数中获得。

为了构造内联固有函数的参数和返回值。首先，必须利用同构映射将模板中的基本块编号转化成目标函数中的基本块编号，由于 T1 和 B2 相匹配，而 T6 和 B7 相匹配，因此 param_filter 变成 $\{ \langle 2, 1, 2 \rangle, \langle 2, 2, 2 \rangle \}$ ， return_filter 变成 $\{ \langle 7, 2, 1 \rangle \}$ 。接着，用新的 param_filter 和 return_filter 索引目标函数。最后，得到 $\text{param_filter} = \{ [eax], [ecx] \}$ ， $\text{return_filter} = \{ eax \}$ 。因此，内联固有函数的参数是 $\{ [eax], [ecx] \}$ ，返回值是 $\{ eax \}$ 。最终，经过固有函数消减之后，目标函数变成图 3-10(c)所示结果。

图同构在软件印迹分析，软件取证，软件克隆检测，软件版本研究等领域有着广泛的应用空间。

3.4.3 基于图拓扑嵌入的内联固有函数消除

在分析现实世界的可执行程序时，往往因为编译器未知和目标平台未知等，导致对内联函数的表现方式不是非常清楚。例如，同样一段包含 `memcmp` 函数的 C 代码在 GCC 编译器中经过内联优化之后的编译结果与在 Visual Studio 编译器在经过内联优化之后的编译结果往往是不相同的；另外，对于同一个编译器在内联固有函数时，当固有函数所处的上下文（如函数参数等）不同时，固有函数的内联策略也是不相同的；最后，对于同一个编译器，即使对于同一个程序，由于编译优化以及编译优化次序的不同，编译的结果也千差万别。

总之，编译器未知，目标平台未知，固有函数上下文未知以及编译器优化及优化次序不同，导致固有函数的内联变化万千，无形中增加了基于子图同构的内联固有函数消除算法的难度，如下面的例子。

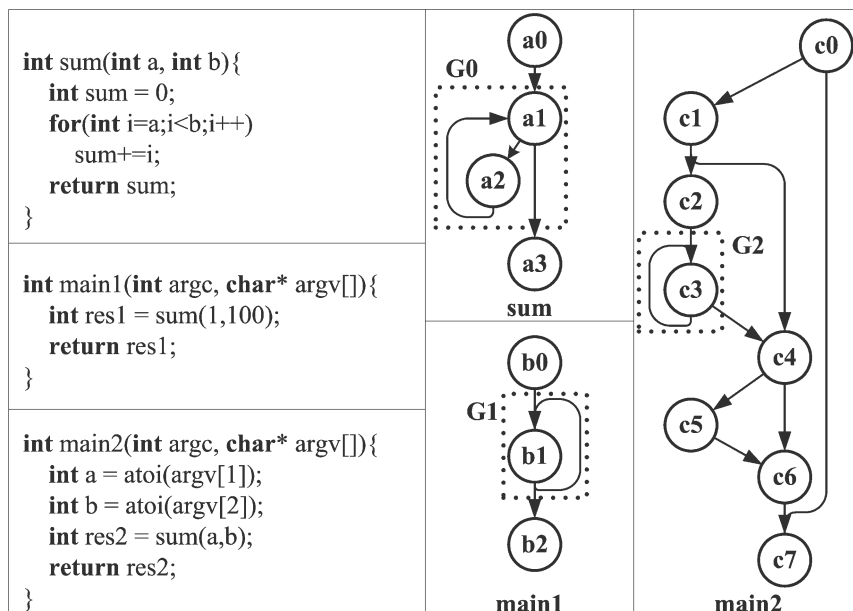


图 3-11 上下文差异对于内联函数展开的影响

图 3-11 列出了 `sum` 函数，`main1` 函数以及 `main2` 函数的源码，同时列出了 `sum` 函数，`main1` 函数以及 `main2` 函数在 Visual Studio C++ 编译器编译后经 IDA 反汇编得到的控制流图。`main1` 函数和 `main2` 函数源码的主要区别在于调用参数不一样，`main1` 函数传递的是常量参数，而 `main2` 函数传递的是未知参数。最终，经过编译器的优化之后，二者的控制流图差别非常大，`main1` 几乎是 `main2` 的一个子图。如果按照子图同构的方法在 `main1` 和 `main2` 上检测 `sum`，是得不到任何结果的。

但是，当采用图拓扑嵌入的判断方法进行内联函数检测时，以子图 $G0 = (\{a1, a2\}, \{\langle a1, a2 \rangle, \langle a2, a1 \rangle\})$ 为模板，子图 $G1 = (\{b1\}, \{\langle b1, b1 \rangle\})$ 和子图 $G2 = (\{c3\}, \{\langle c3, c3 \rangle\})$ 为目标对象，那么由于子图 $G1$ 和子图 $G2$ 都拓扑嵌入到子图 $G0$ 中，因此子图 $G1$ 和子图 $G2$ 可以成为候选的内联函数集合。然后，再通过基本块语义的判断，确定真正的内联函数。

3.5 本章小结

本章介绍了反编译器框架和程序结构优化方面的算法。反编译器框架设计是在分析了 2.1 节中各个反编译器的架构及其缺陷的基础上,提出了智能解码器以及提高兼容性的中间语言生成模块。

在程序结构优化方面,针对编译器对 `switch` 结构的不同处理方式,提出了基于跳转表的 `switch` 结构的优化算法和基于二叉树的 `switch` 结构的优化算法。`Goto` 语句是反编译结果中最影响阅读和理解的因素,而其引入原因又是多种多样的,它是反编译优化的综合效果,通过代码复制的方法消除 `goto` 语句是一种折中的方法。

基于子图同构的内联固有函数的消除算法首先利用子图同构算法检测目标函数中的固有函数,然后利用固有函数以及目标函数的映射关系,找到对应的内联固有函数的参数和返回值。基于子图同构的内联固有函数消除算法在检测中存在一些问题,例如编译优化会在内联固有函数中插入额外的基本块,进而影响子图同构识别算法的分析。而图拓扑嵌入可以很好的解决此类问题,这也是以后工作的重心之一。

第 3.4 节虽然为内联固有函数的消除算法研究,但是,子图同构和图拓扑嵌入可以用在其它相关领域的研究,例如被内联的用户自定义函数的消除。在这种情况下,首先,需要在可执行程序中分析出复杂度比较低的用户自定义函数,因为这些函数很有可能被编译器内联到代码中。然后到剩余函数集合中去匹配,并将相应的子图进行规约,提高反编译结果的抽象程度和可读性。

4 ASMBoom 反编译器实现

反编译器的处理对象可以是二进制代码，汇编程序，JAVA 字节码，MSIL，LLVM IR 等任何一种机器代码或中间代码。本章的研究重点是面向汇编程序的反编译器 ASMBoom 的实现以及智能解码器中构造各种模板库的过程。

4.1 反编译器实现

本文中的 ASMBoom 反编译器的输入是汇编程序，输出是类 C 程序。其系统框架如下：

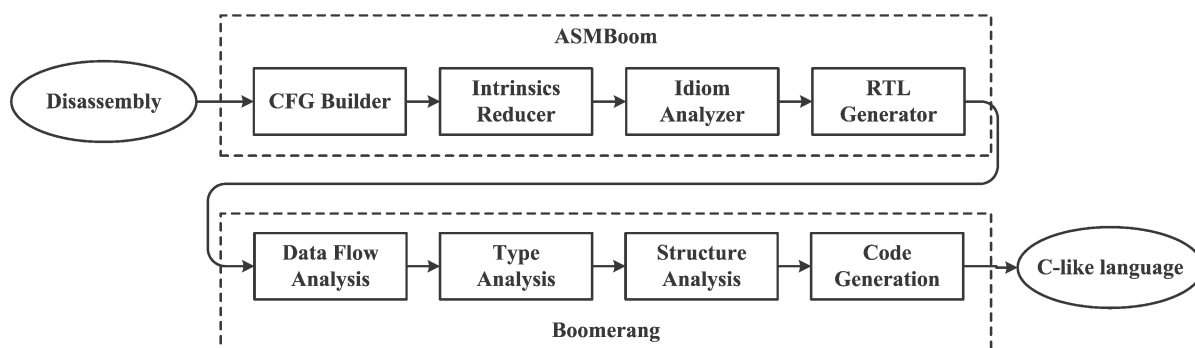


图 4-1 ASMBoom 反编译系统框架图

图 4-1 展示了 ASMBoom 反编译器的框架图，它由两部分组成，ASMBoom 和 Boomerang。ASMBoom 构成了整个反编译器的前端，其主要负责控制流图的恢复（CFG Builder），固有函数消除（Intrinsics Reducer），指令习语分析（Idiom Analyzer）和 RTL 生成（RTL Generator）。ASMBoom 反编译器的后端采用 Boomerang 反编译器中的数据流分析，类型分析，结构化分析和代码生成。

CFG Builder 将顺序的汇编程序转换成基于控制流图的程序表达方式。在控制流图的恢复过程中，最大的挑战是间接跳转地址的确定，跳转表的翻译和间接函数调用的确定。目前，在控制流图恢复方面做得比较好的是 VSA 算法^[27]和 Jakstab 反汇编器中所使用的结合数据流分析的控制流图构建算法^[19]。在 ASMBoom 系统中，CFG Recovery 算法在间接跳转方面，只处理了基于跳转表的 switch 结构的恢复，对于其它间接跳转和间接函数调用没有做特殊处理。因此，当被反编译程序中出现这些结构时，ASMBoom 反编译器构造出来的控制流图可能不完整。

Intrinsics Reducer 在基于控制流图的汇编程序中检测内联的固有函数，并根据内联函数使用参数和构造返回值的模式，恢复出参数和返回值，从而将内联的固有函数消减成一个简单的函数调用，简化控制流图，提高后续的数据流分析和控制流分析的精确度；丰富基本类型信息，简化类型分析复杂度和提高后续类型分析的精度。

Idiom Analyzer 在程序控制流图中的基本块中分析常见的指令习语序列，并将这些

指令习语序列翻译成抽象程度更高，可读性更高的操作。最常见的指令习语如除以非 2^n 次方整数，对非 2^n 次方整数取余等操作，在抽象这些指令序列流时，需要通过魔幻数和移位数等来恢复相应的被除数，详细内容参考^[91]。

RTL Generator 遍历控制流图中每个基本块中的汇编指令，并利用指令操作码去 RTL 词典中查询相应指令的中间表达式，完成参数替换之后，得到该指令的中间表达式。最终，整个控制流图中的指令被转换成中间表达式，完成整个程序到中间表达式的转换工作。

Data Flow Analysis 通过活跃变量分析来消除无用代码，确定被调用过程体的参数和返回值等；通过到达定值分析，进行表达式传播和表达式组合等。

Type Analysis 从机器指令的操作码，库函数签名和常数等处获取基本类型信息，然后利用类型推导规则推导其它变量的基本类型，从而使得生成的高级代码的可读性更强。

Structure Analysis 根据结构化算法，将控制流图中的结点按照其在控制流图中的位置分成不同的类别，如顺序代码块、分支代码块和循环代码块等。**Goto** 语句的消除是在 **Structure Analysis** 之前完成，从而保证变量和表达式的一致性。

Code Generation 通过遍历控制流图，依据每个基本块所属高级结构的类型，分别生成顺序、分支和循环的代码。整个代码生成过程是个自顶向下，从程序到过程体，到控制流图，到基本块，到语句，到表达式，逐层深入的过程。

当然，为了进一步提高程序的可读性，可能要对某些地址值进行重新命名，例如将全局地址命名成带有 **global** 前缀的名字，将局部地址命名为带有 **local** 前缀的名字，将参数命名成带有 **var** 前缀的名字等；为了回溯汇编代码，以便进行跟踪比对，可以在高级程序的分支或循环结构处、特殊操作处标注汇编指令的地址范围等；在输出 **switch** 结构时，需要将跳往同一基本块的多个 **case** 分支进行合并，从而有效减少不必要的 **goto** 语句。

4.1.1 Boomerang 反编译器

Boomerang 是一款开源的二进制程序的反编译器，支持 **x86**, **ppc**, **mips** 和 **arm** 等多种指令系统，同时还支持 **ELF**, **PE** 等多种文件格式。虽然，**Boomerang** 反编译器自 2007 年开源以后，其开发以及维护不再活跃。但是，**Boomerang** 反编译器对于反编译的影响是有目共睹的。本节将从软件工程的角度来分析 **Boomerang** 反编译器的设计和实现。突出 **Boomerang** 反编译器对于程序的建模，从最高层的程序，到过程体，到控制流图，到基本块，到语句，再到表达式。**Boomerang** 反编译器中程序组成元素以及相关建模类的对应关系如表 4-1。

表 4-1 分析了 **Boomerang** 反编译器中程序自顶向下的组织结构，从最高层的文件，程序到最低层的语句，表达式和类型。除此之外，还有许多重要的类用来辅助反编译流程，其中最重要的一个类是 **DataFlow** 类，该类是 **Boomerang** 反编译器进行 **SSA** 分析的主要类，例如计算支配结点，直接支配结点，活跃变量，插入 **Phi** 结点，变量重

命名等都由该类完成。

表 4-1 Boomerang 中程序组织相关类及功能说明

程序组成	相关类	类功能说明
文件模块	BinaryFile 类及派生类	描述不同文件的组织和内容，存储文件的文件头，程序头表，节，节头表和二进制代码等信息
程序	Prog 类，Global 类	二进制程序的最高组织结构，存储整个程序，二进制文件内容以及全局变量等信息
过程体	Proc 类，Parameter 类，Return 类，Signature 类	反编译优化的主要单位，如参数和返回值分析，数据流分析，类型分析，控制流分析，代码生成等
控制流图	Cfg 类	反编译优化的实体，包括支配结点计算，活跃变量分析，到达定值计算，结构规整，结构化算法等
基本块	BasicBlock 类	连续代码块的存储单元，用于局部表达式组合，表达式化简，类窥孔优化等
语句	Statement 类及派生类	建模程序中的跳转语句，分支语句，函数调用语句，赋值语句，返回语句，Phi 语句等
表达式	Exp 类及派生类	建模程序中的一元，二元和三元运算，常量，内存变量，标志位，类型子表达式，指针表达式等
类型	Type 类及派生类	建模程序中的 void，函数类型，bool，char，integer，float，pointer，array，union 等类型

有些时候，SSL 描述指令语义的能力有限，这时候需要在反编译软件的解码器中进行指令解码以及中间代码的构造，具体的例子如 ppc 中的 lmw 和 stmw 指令。在 Boomerang 反编译器自带的 ppc 指令系统的语义描述文件中，其描述如下表。

表 4-2 Boomerang 对 ppc 中 lmw 和 stmw 指令的语义描述示意

lmw rd, eaddr, d	stmw rs, eaddr, s
tmp := 0	tmp := 0
d <= 0 => r[d + tmp] := m[eaddr + (tmp*4)]	s <= 0 => m[eaddr + (tmp*4)] := r[s + tmp]
d <= 0 => tmp := tmp + 1	s <= 0 => tmp := tmp + 1
d <= 1 => r[d + tmp] := m[eaddr + (tmp*4)]	s <= 1 => m[eaddr + (tmp*4)] := r[s + tmp]
d <= 1 => tmp := tmp + 1	s <= 1 => tmp := tmp + 1
.....
d <= 30 => r[d + tmp] := m[eaddr + (tmp*4)]	s <= 30 => m[eaddr + (tmp*4)] := r[s + tmp]
d <= 30 => tmp := tmp + 1	s <= 30 => tmp := tmp + 1
d <= 31 => r[d + tmp] := m[eaddr + (tmp*4)];	s <= 31 => m[eaddr + (tmp*4)] := r[s + tmp];

表 4-2 列出了 lmw 和 stmw 指令的 SSL 描述情况，为了示意的简洁，中间省略了 56 行的描述，分别是卫士表达式从 d <= 2 到 d <= 29 的描述。表 4-2 中的 d <= 0 是一个卫士表达式，r[·] 表示第 · 个寄存器，m[·] 表示地址为 · 的内存。从 SSL 的描述中可以看出，lmw rd, eaddr, d 表示从连续内存地址 eaddr 中读取数值，并将其赋值给编号从 d 到 31 的通用寄存器，在 ppc 体系结构中负责函数调用栈中函数返回值的回传。stmw

rs, eaddr, s 表示将编号从 s 到 31 的通用寄存器复制给连续内存地址 eaddr 中, 负责函数调用栈中参数的初始化。

Boomerang 反编译器的控制流图构建和中间代码生成是同步完成的, 采用递归扫描的方式构建, 初始时用函数入口地址初始化未访问地址链表。每次从未访问地址链表中取出头地址, 如果该地址处的指令已经解码, 则分裂包含该地址的基本块; 否则, 解码从该地址开始的连续指令, 直到遇到终结指令 (如函数调用, 跳转指令, 间接跳转指令, 函数返回指令等); 接着, 根据基本块结尾处的指令类型, 分别构造不同种类的基本块 (如函数调用基本块, 二分支基本块, 多分支基本块, 结束基本块等), 并将相应基本块的后续地址和跳转地址依次加入未访问地址链表中; 重复执行以上步骤, 直至未访问地址链表为空, 结束。

Boomerang 反编译器中的 SSA 分析是 Van Emmerik 在其博士论文中提出并实现的。Boomerang 反编译器借助 SSA 来辅助表达式传播, 参数和返回值的确定, 死代码删除等。SSA 分析的关键步骤是: (1) 确定 Φ 函数的插入点; (2) 系统地将所有变量重命名。常见的 SSA 构造算法有 Minimal SSA, Pruned SSA 和 Semi-pruned SSA。三者的关键区别在于确定 Φ 函数的插入点问题上。记变量 v 的所有定义基本块集合为 $X(v)$, 基本块 B 的支配边界为 $DF(B)$, 基本块 B 入口处的活跃变量集合为 $LIVE_IN(B)$, 基本块 B 中使用的非局部变量集合为 $NON_LOCAL(B)$ 。

(1) Minimal SSA 算法在支配边界的基本块中插入 Φ 函数, 即对于程序中的每个变量 v , 其 Φ 函数插入基本块集合为 $\phi(v) = \{a \mid \forall b \in X(v), \forall a \in DF(b) \rightarrow a\}$ 。

(2) Pruned SSA 算法为了减少 Φ 函数的数目, 不仅考虑支配边界, 同时还考虑变量的活跃性, 其只为支配边界上后续活跃的变量插入 Φ 函数。其 Φ 函数插入基本块集合为 $\phi(v) = \{a \mid \forall b \in X(v), \forall a = DF(b), v \in LIVE_IN(a) \rightarrow a\}$ 。

(3) Semi-pruned SSA 算法在分析了 Pruned SSA 算法在计算活跃变量的复杂度较高的情形下, 提出了非局部变量, 将活跃变量的计算转换成非局部变量的计算。其 Φ 函数插入基本块集合为 $\phi(v) = \{a \mid \forall b \in X(v), \forall a = DF(b), v \in NON_LOCAL(b) \rightarrow a\}$ 。

在系统地进行变量重命名时, 通常引入两个数据结构, 一个是 $Stacks[v]$, 用来存储当前变量 v 的下标; 另一个是 $Counters[v]$, 用来存储变量 v 的下一个下标。变量重命名模块在递归遍历整个控制流图的支配树时, $Stacks$ 和 $Counters$ 被动态维护。在当前基本块中, 对于变量 v 的每一次定义, 变量重命名模块会用 $Counters[v]$ 将 v 重命名为 $v_{Counters[v]}$, 并将 $Counters[v]$ 压入到栈 $Stacks[v]$ 上, 同时增加 $Counters[v]$ 。对于当前基本块在控制流图上的每一个后继基本块, 对在 Φ 函数中使用的变量 v , 都会用 $v_{top(Stacks[v])}$ 来重命名变量 v 。变量重命名模块递归的重命名当前基本块在支配树上的每一个后继基本块。在当前基本块及其在支配树上的后继结点集合被处理完之后, 要对 $Stacks[v]$ 进行恢复, 使其恢复到进入当前基本块前的情形。

以上的 SSA 分析对于将基于控制流图的汇编程序转换成 SSA 形式的 LLVM IR 中间表达式具有重要意义。

4.1.2 ASMBoom 前端工作流程

本节从 ASMBoom 前端的工作细节入手，说明程序在 ASMBoom 前端的变化过程以及 ASMBoom 的前端输出。

图 4-2 展示的 ASMBoom 前端包括加载器 (Loader)，控制流图构建器 (CFG Builder)，固有函数消减和指令习语分析器 (Intrinsics Reducer & Idiom Analyzer)，SSL 解析器 (SSL Parser) 和中间代码生成器 (IR Generator) 等五个部分。

Loader 根据汇编语言文法，将汇编程序读入到反编译程序中，并且组织成相应的数据结构，如符号表，符号地址表，函数地址表，跳转链表，指令链表 asmProg 等。

CFG Builder 根据 PC 值，从 asmProg 中获取一条指令并存储到 asmInstr 中。接着，根据指令 asmInstr 的类型去更新 PC 的值，如果是顺序指令，则将 PC 修改成 $PC + InstrLength(asmInstr)$ ，其中 $InstrLength(asmInstr)$ 为指令 asmInstr 的长度；否则，解码器要根据符号地址去符号地址表中查询其具体地址 tAddr，并将其赋给 PC 作为下一次取指令的地址。最后，根据控制流重构算法^[65]实时地构造出汇编程序的控制结构。通过上述三步，最终将平坦的汇编指令序列转换成具有控制结构的汇编程序。Switch 结构的构建和恢复也是在该阶段完成。

Intrinsics Reducer 根据算法 3-4 在基于控制流图的汇编程序上检测内联固有函数，并识别其参数和返回值。Idiom Analyzer 根据算法 3-1 抽象基本块中的指令习语，将其转换成可读性更高的伪汇编指令。Intrinsics Reducer & Idiom Analyzer 部分将 Structured asm prog 转换成了 Lifted asm prog。

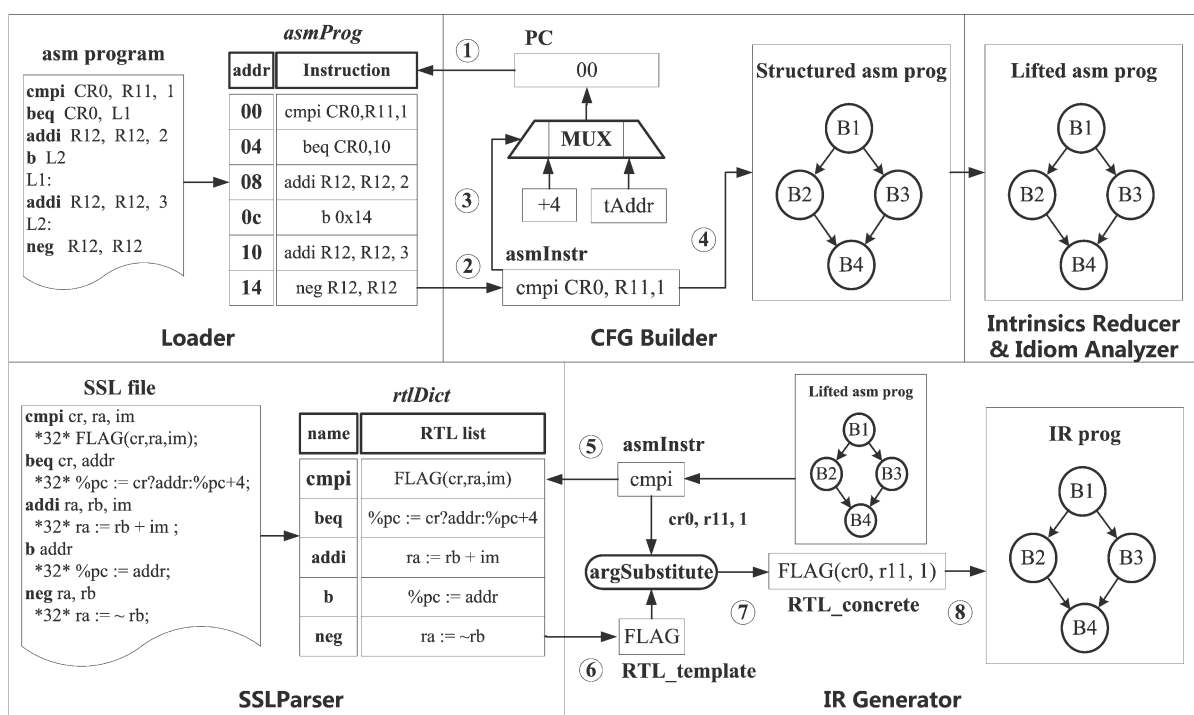


图 4-2 ASMBoom 前端详细工作流程

SSLParser 通过对描述指令系统语义的 SSL file 进行词法和语法分析之后，构造出

通用寄存器链表，标志寄存器链表，中间语言词典 `rtlDict` 等。由于在智能解码器中进行了 `Intrinsics Reducer` 和 `Idiom Analyzer` 等抽象操作，因此，此处的 `SSL file` 通常描述的是扩展之后的指令系统的语义。

`IR Generator` 遍历 `Lifted asm prog` 中的每一个基本块中的每一条指令，提取并分离操作码以及操作数，利用操作码索引中间语言词典 `rtlDict` 中相应指令的 `RTL_template`。参数替换模块 `argSubstitute` 将 `RTL_template` 和操作数列表，组装成 `RTL_concrete`，并最终将其插入到对应的汇编指令的位置处。

`ASMBoom` 反编译器前端的最大特点是在基于控制流图的汇编程序上进行内联固有函数的消除和指令习语的分析，而二者的关键是相应模板库的构建，模板库的完整性和正确性在很大程度上决定着中间表达式的正确性。关于模板库的构建内容将在第 4.3 节展开介绍。

4.1.3 可扩展的前端

反编译器的前端的变化因素主要有两个，一个是文件类型，另一个是指令系统。设有 m 种文件类型， n 种指令系统，那么总共要实现 $m \times n$ 种前端。但是为了提高软件复用率，将前端分解为文件类型和指令系统两大模块。那么对于同样的需求，只需要写 m 种文件解析单元和 n 种指令系统解析单元。在参考了 `Boomerang` 反编译器的设计方案之后，最终的基于汇编语言的反编译器 `ASMBoom` 软件的实现如下。

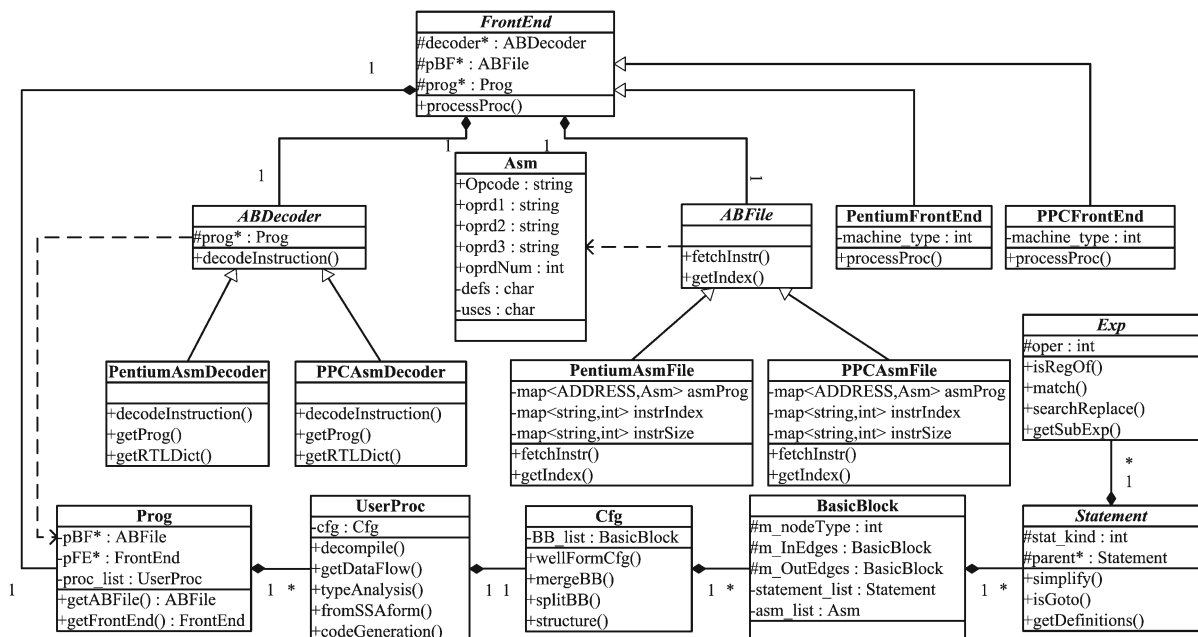


图 4-3 ASMBoom 反编译器的主要类图

图 4-3 中列举了 `ASMBoom` 反编译器中前端的主要类以及程序在反编译器的组织相关的类（为了展示清楚，类中只列出了重要的属性和方法）。最上面的是 `FrontEnd` 类，其派生出 `PentiumFrontEnd` 类和 `PPCFrontEnd` 类。`FrontEnd` 类中最重要的三个组成部分是 `ABFile` 类，`ABDecoder` 类和 `Prog` 类。`ABFile` 类负责二进制或者汇编程序的存储；`ABDecoder` 类负责二进制程序或汇编程序的解码，内联固有函数消除和指令习语

分析等；Prog 类负责整个程序的存储，其由一系列过程体 UserProc 组成，而 UserProc 的核心是 Cfg 类，即程序的控制流图。程序的控制流图 Cfg 由基本块链表组成，其中入度为 0 的基本块是程序的开始基本块，出度为 0 的基本块是程序的结束基本块。基本块 BasicBlock 类，根据 m_nodeType 属性，被分成不同种类，常见的基本块类型有函数调用基本块，二分支基本块，多分支基本块，结束基本块等。每个基本块既存储着指令的中间表达式还存储着原始的汇编指令，分别被存储在 statement_list 链表和 asm_list 中。同时，每个基本块的出边和入边都被存储，以方便控制流分析。过程体中的语句用 Statement 类来建模，而每条语句又由许多表达式 Exp 类组成。

前端的可扩展性主要体现在 FrontEnd 类的设计，该类可以派生出任何文件类型和任何指令系统的可执行程序。只需要从 ABFile 类中派生相应文件类型的解析类以及从 ABDecoder 类派生相应指令系统的解码类即可。在图 4-3 中，从 ABFile 类派生出了 PentiumAsmFile 类和 PPCAsmFile 类，分别完成 Pentium 汇编文件的解析和 ppc 汇编文件的解析，解析的指令最终存储在 asmProg 字典中；从 ABDecoder 类派生出了 PentiumAsmDecoder 类和 PPCAsmDecoder 类，分别完成 Pentium 指令系统的解码和 ppc 指令系统的解码，通过 decodeInstruction 方法完成。由于 ASMBloom 反编译器是一款面向汇编程序的反编译器，因此将 PentiumAsmFile 类与 PentiumAsmDecoder 类组合成 Pentium 前端 PentiumFrontEnd，将 PPCAsmFile 类与 PPCAsmDecoder 类组合成 ppc 前端 PPCFrontEnd。

汇编指令被存储在 Asm 类中，其包括操作码，操作数，定义和使用。汇编指令的定义和使用可以用在算法 3-3 中，构建指令依赖图，对指令进行聚类，消除指令调度对指令习语的影响，进而检测和恢复更多的指令习语。

在 Prog 类中通过 decodeInstruction()方法解码每条指令时，需要首先通过方法 getABFile()获得其抽象类 ABFile 的实例；然后，调用 fetchInstr()方法，根据多态性，ABFile 会有根据地调用相应子类的 fetchInstr()方法，进而实现具体的取指令功能。

基于图 4-3 的前端设计，在实际开发中，已经成功的指导了基于 mips 和 arm 等汇编程序的反编译器模型的实现。

4.2 固有函数消除算法实现

在基于子图同构的内联固有函数消除算法中，主要包括固有函数模板库的构建，子图同构算法设计，基本块语义的兼容性判断以及固有函数参数和返回值的分析。本节着重介绍固有函数模板的构造以及 vflib 软件的介绍和使用说明。

4.2.1 固有函数模板库构造

固有函数是写在编译器中的代码，而且以编译器中间代码写成，因此很难通过直接的方式获取编译器中的所有固有函数模板。构造模板库只能通过间接的方式获得，本节就构造简单模板库的方法做一个说明。

首先，构造包含不同固有函数的简单测试集 A。集合 A 中的每个测试程序只包含

一个 main 过程体。集合 A 中的某些测试集调用的固有函数是重复的，但是其参数的种类是有所差别的。根据分析，传递常数参数的固有函数与传递变量参数的固有函数在内联编译优化过程中的处理流程是不一样的。对于变量参数的固有函数的优化要比常数参数的固有函数的优化更复杂一些，最直观的例子，如图 3-11 的 main1 函数和 main2 函数的控制流图对比。

其次，将测试集 A 用不同的编译器，不同的优化组合去编译得到可执行程序集合 $B = \{b_{11}, b_{12}, \dots, b_{1n}, b_{21}, b_{22}, \dots, b_{2n}, \dots, b_{m1}, b_{m2}, \dots, b_{mn}\}$ ，其中 b_{ij} 表示第 i 种编译器利用第 j 种编译优化算法得到的可执行程序。这里，为了实现固有函数的内联，编译优化经常是 -O2 以上的优化。

再次，利用反汇编器反汇编可执行程序集合，并存储每一个 main 过程体。其存储内容即包括 main 过程体的控制流图，同时还包括控制流图中每个基本块中的汇编指令操作码序列。最终构成汇编程序集合 $Asm = \{c_{11}, c_{12}, \dots, c_{1n}, c_{21}, c_{22}, \dots, c_{2n}, \dots, c_{m1}, c_{m2}, \dots, c_{mn}\}$ 。

最后，利用图同构的方法，对汇编程序集合 Asm 进行聚类。聚类完成之后，只保留各个类的代表元作为测试集合，记为 P。通过聚类并删除非代表元的方法可以减小固有函数模板库的大小，提高子图同构匹配的效率。

经过以上四步之后，一个完整的固有函数模板库被构造起来。在 ASMBoom 系统里，目前已经包含一个支持如下固有函数集的模板库。

表 4-3 固有函数列表及功能介绍

固有函数名称	函数描述
strlen	计算字符串长度
strcpy	将指定长度的字符串从源地址拷贝到目标地址
memset	设置固定长度的内存块为指定值
strcmp	比较两个字符串，相等则返回 0；否则，其它
memcmp	比较指定长度的两块内存的内容

表 4-3 列举的固有函数都是字符串拷贝的函数，根据表 3-2 分析可知，固有函数的种类繁多。在实际中，可以根据反编译的对象的特点，构造不同的固有函数模板库。例如，在嵌入式系统中，由于其计算资源的有限，往往通过固有函数来模拟相应的高级操作，最常见的是用固有函数在 32 位嵌入式平台上模拟 64 位乘除法的运算操作。

4.2.2 Vflib 库介绍

Vflib 是萨勒诺大学开发的 MIVIA 实验室开发的用来计算图匹配的 C++ 库。被广泛地使用在虚拟网络映射，SoC 设计，RDF 索引，生物网络，分子检测，客户行为挖掘和图索引等领域。

图 4-4 展示了整个 vflib 库中最重要的类以及类之间的继承和依赖关系。整个图被存储在 Graph 类中，类中存储着结点数目，结点属性，边属性，每个结点的入边数目和出边数目以及属性比较器等。但是，Graph 并不直接暴露给用户使用，通常通过 ARGraph 类去访问图中的一些属性。ARGraph 往往通过 ARGLoader 类来构造，ARGraph

派生出 ARGEdit 类，完成图的编辑，例如图的结点和边的插入和删除等。从 ARGEdit 类派生出的两个类 BinaryARGLoader 可以从二进制文件中加载图或者将图写入到二进制文件中，因此该类中的方法通常是 readWord 和 writeWord；StreamARGLoader 类从文本文件中读取一个图或者将图写入到文本文件中，其方法包括 readLine，用于读取文件中的一行内容；readCount 用来读取图中结点数目；readNode 读取结点信息；readEdge 读取边集的信息。

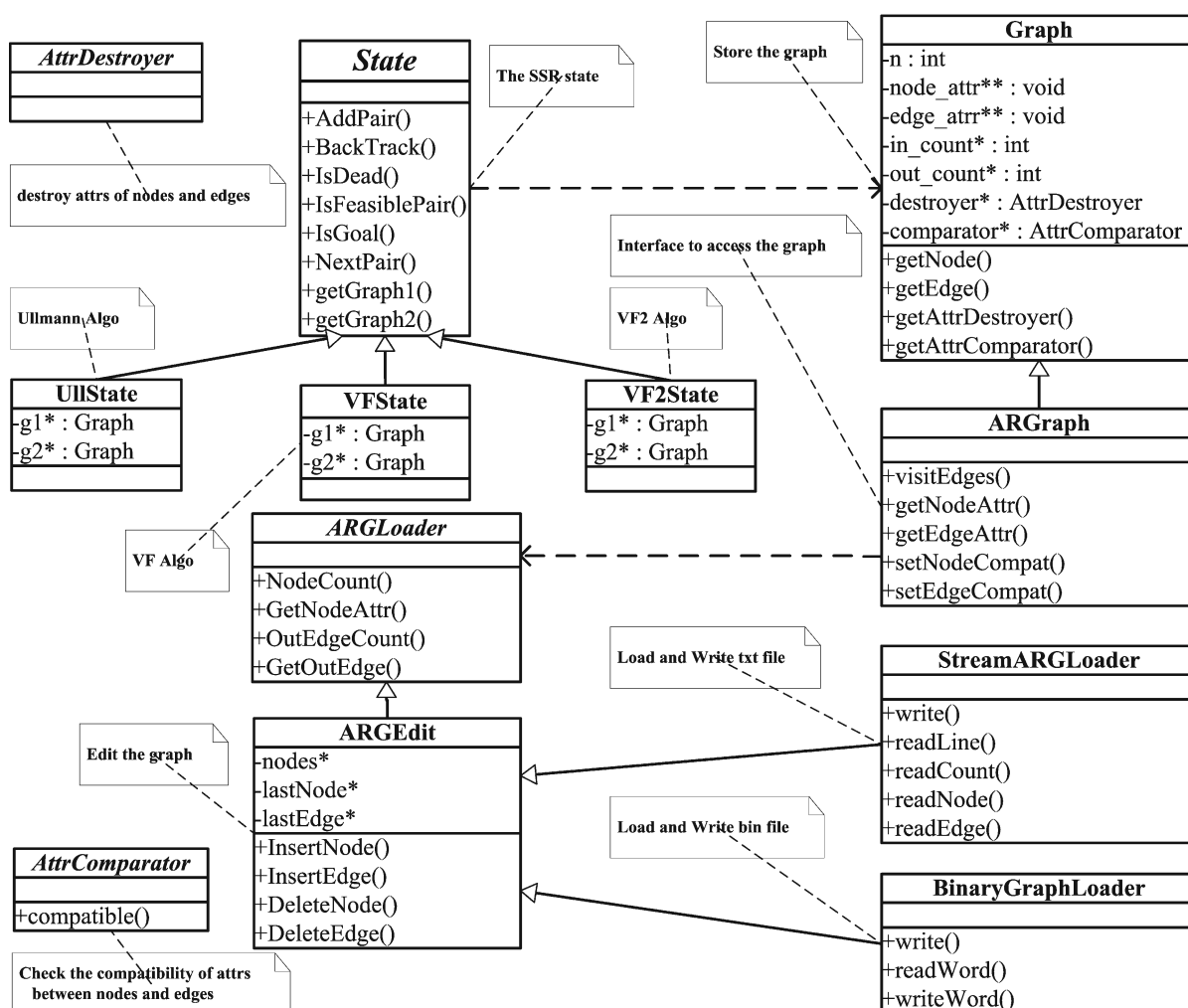


图 4-4 vplib 类图说明

AttrDestroyer 类完成图中结点和边集上属性的删除。AttrComparator 类用来设置图中结点和边集的兼容性比较策略。

Vflib 中最重要的是 State 及其派生类，为了简化类图，图 4-4 只绘出了三个主要派生类：UllState，VFState 和 VF2State 类，分别实现 Ullmann 算法，VF 算法和 VF2 算法。Vflib 中的图匹配算法都是基于状态空间搜索匹配的，State 及其派生类根据算法的不同，采用不同的状态空间维护策略。AddPair 用来增加候选状态，NextPair 用于返回下一个状态，IsFeasiblePair 用于判断某一状态是不是合适的选择，从而确定是否对该搜索分支进行剪枝，IsGoal 判断是否完成了匹配，BackTrack 在匹配不成功返回时，恢复遍历中的一些数据，如映射集合等。

在将 vflib 引用到 ASMBoom 反编译器中时，最重要的是将 ASMBoom 中的 Cfg 类表示的控制流图的拓扑结构以及相关的结点和边的属性映射到 ARGEdit 类中。在 ASMBoom 系统中，结点映射以及对应的拓扑结构的映射都可以通过遍历控制流图完成，如下表所示。

表 4-4 vflib 使用示例伪代码

```

01. //输入:: 程序控制流图Cfg; 模板库templates。
02. ARGEdit target;
03. int bbNum = 0;
04. std::map<BasicBlock*, int> bbMap;    // 基本块及其在target中的编号映射表
05. // 插入结点
06. for each BasicBlock B in Cfg
07.     target.InsertNode(B);    //将基本块B插入到target中作为结点
08.     bbMap.insert(new Pair<BasicBlock*, int>(B, bbNum++)); //将B及其编号插入到bbMap
09. end for
10. // 插入边
11. for each edge E in Cfg
12.     int srcNum = bbMap.find(E.src);
13.     int dstNum = bbMap.find(E.dst);
14.     target.InsertEdge(srcNum, dstNum); // 将边<srcNum, dstNum>插入到target中
15. end for
16. // 构造ARGraph
17. ARGraph<BasicBlock, void> g1(&target);
18. g1.SetNodeComparator(new BasicBlockComparator(void*, void*)); // 设置结点属性比较器
19. // 遍历模板库
20. for each g2 in templates
21.     VF2SubState s0(&g2,&g1);    // 利用VF2算法进行子图同构匹配
22.     node_id ni1[100], ni2[100];
23.     if !match(&s0,&bbNum,ni1,ni2)
24.         // Replace and Substitute.....
25.     end if
26. end for

```

表 4-4 展示了利用 vflib 判断模板库 templates 中在 Cfg 中是否有相匹配的子图。首先，申明一个 ARGEdit 对象 target，将 Cfg 中的结点插入到 target 中，并对结点进行编号。其次，遍历控制流图 Cfg 中的边集，将其插入到 ARGEdit 中。完成了 target 的构造之后，需要将 target 转换成图的格式，因此构造了 ARGraph 类的对象 g1。再次，设置基本块属性兼容性判断的方法，在 ASMBoom 反编译器中使用的是 BasicBlockComparator 方法，其接收两个 void 指针，在传入过程体之后，被转换成汇编指令链表，然后根据 3.4.2 节中的方法判断两个基本块的语义兼容性。最后，程序遍历模板库 templates 中的每一个模板 g2，并将模板 g2 与 g1 利用 VF2 算法进行 match。如果匹配，则将 Cfg 中对应的控制流子图进行删除，代之以函数调用，并利用函数参数和返回值识别技术恢复被调用函数的参数。

通过表 4-4 的例子可以看出，vflib 的使用简单方便，而且效率也非常高。

4.3 指令习语数据库构建

在实际分析程序中，常面临编译器未知，目标平台未知，编译优化及其优化顺序未知等问题。这些问题导致指令习语的差别非常大。本节旨在说明如何通过构造指令序列模板库，并用频繁顺序模式的数据挖掘手段分析程序中的指令习语，并将这些指令习语用在智能解码器上，指导 ASMBoom 前端很好的解码并翻译指令习语，并将这些指令习语进行抽象，提高可读性。在程序样本特定的情况下，指令习语通常是那些频繁的指令序列。

指令习语通常是编译器为了提高程序执行速度或者是减小程序的体积而采用的代码段等价替换的方法。本节所讲的指令习语假设都集中在一个基本块范围内。指令习语模板库的构造方法如下。

首先，选择不同种类的样本构成样本集 $A = \{a_1, a_2, \dots, a_n\}$ ，每个 a_i 表示一种类型的样本集，例如可以表示除以 2 的整次幂，乘以 2 的整次幂，除以非 2 的整次幂，对非 2 的整次幂取模，数组访问，链表操作等程序。将样本集 $A = \{a_1, a_2, \dots, a_n\}$ 用不同种类的编译器集合的不同优化遍及编译优化组合 $C = \{c_1, c_2, \dots, c_m\}$ 编译。最终，编译得到可执行程序集合 $B = \{b_{11}, b_{12}, \dots, b_{1n}, b_{21}, b_{22}, \dots, b_{2n}, \dots, b_{m1}, b_{m2}, \dots, b_{mn}\}$ ，其中 b_{ij} 表示第 i 种编译器的优化遍及编译优化组合对样本 j 的编译优化结果。

其次，反汇编可执行程序集合 $B = \{b_{11}, b_{12}, \dots, b_{1n}, b_{21}, b_{22}, \dots, b_{2n}, \dots, b_{m1}, b_{m2}, \dots, b_{mn}\}$ ，最终得到汇编程序集合 $Asm = \{d_{11}, d_{12}, \dots, d_{1n}, d_{21}, d_{22}, \dots, d_{2n}, \dots, d_{m1}, d_{m2}, \dots, d_{mn}\}$ 。对于集合 Asm 中的每一个汇编程序 d_{ij} ，利用反编译器 ASMBoom 构造其控制流图 CFG_{ij} 。为 CFG_{ij} 中的每一个基本块构建程序切片集合 $SLICE_{ij}^k = \{S_{ij}^k(1), S_{ij}^k(2), \dots, S_{ij}^k(l)\}$ ，其中 $SLICE_{ij}^k$ 的上标 k 表示 CFG_{ij} 中的第 k 个基本块， $S_{ij}^k(g), g \in \{1, 2, \dots, l\}$ 表示 CFG_{ij} 中的第 k 个基本块的第 g 个程序切片。将 CFG_{ij} 的每个程序切片看成一个事务，最终组合成一个事务集合 $T_{ij} = \bigcup_{b \in CFG_{ij}} SLICE_{ij}^b$ ，利用频繁顺序

模式挖掘手段挖掘程序中的频繁顺序序列，最终构成指令习语 $\langle i, j, idiom \rangle$ ，表示第 i 种编译器的优化遍及编译优化组合对操作 j 的指令习语， $idiom$ 表示相应的指令习语流。

最后，组合所有的 $\langle i, j, idiom \rangle$ 对，构成指令习语数据库 $IDIOM = \bigcup_{i \in C, j \in A} \langle i, j, idiom \rangle$ 。

在构建事务的时候，除了可以用每个基本块中的程序切片作为基本事务之外，还可以将每个基本块中的指令流作为事务或每条指令为一个基本事务。为了消除不相关事务对频繁顺序模式挖掘的影响，在构建基本事务的时候可以做一些初筛，例如在分析除以非 2 的整次幂的指令习语时，可以忽略控制流中以访存为主的基本块，反之亦然。

根据实验，最好的事务构成应当是，以基本块的编号作为事务中的客户编号，以基本块切片的编号作为事务编号，以切片中的指令作为商品。频繁顺序模式挖掘可以采用 Spam 软件^[92]实现。

4.4 Switch 结构优化实现

Switch 结构的优化包括 switch 结构的构造和 switch 结构的规整。对于基于跳转表的 switch 结构在如图 4-1 中的 CFG Builder 阶段完成构造，对于基于二叉树的 switch 结构在 CFG Builder 之后完成构造。Switch 结构的规整则在控制流图构建完成之后，即 CFG Builder 之后完成。本节将介绍如何利用图 4-3 中的数据结构和方法实现 switch 结构的构造和规整。

<pre> // bctr指令解码片段 00. DecodeResult result; 01. ABFile *paf = prog->getABFile(); 02. CaseStatement* newJump = new CaseStatement; 03. Exp* baseAddr = new Const(jumpTableAddr); 04. Asm* prevInstr = paf->fetchInstr(pc - 4); 05. int rd = reg2int(prevInstr->opr1); 06. Exp* relAddr = Location::regOf(rd); 07. Exp *fa = new Binary(opPlus,baseAddr,relAddr); 08. newJump->setDest(fa); 09. SWITCH_INFO *sInfo = new SWITCH_INFO; 10. sInfo->pSwitchVar = relAddr; 11. newJump->setSwitchInfo(sInfo); 12. result.rtl = new RTL(pc, stmts); 13. result.rtl->appendStmt(newJump); 14. result.numBytes = 4; // 结束:: bctr指令解码片段 </pre>	<pre> // pc 为当前解码指令的地址 //00.解码结果, 返给 cfg builder //01.从程序获取文件指针 //02.构造一个多分支语句 //03.构造跳转表基址表达式 /*04.通过多态函数 fetchInstr, 获取分支变量所在指令*/ //06.构造相对地址 relAddr //07.构造全地址 fa //08.设置 newJump 的跳转地址 //09.存储 switch 信息的结构体 //10.设置分支变量为 relAddr //12.在 pc 处构造语句 rtl //13.将 newJump 加入到 rtl 中 //14.设置指令的长度 </pre>
<pre> // 构造多分支控制流结构代码片段 15. Cfg* pCfg = NULL; 16. ABFile* pBF; 17. std::list<RTL*>* BB_rtls; 18. TargetQueue targetQueue.initial(curProc->lowAddr()); 19. Exp* pDest = stmt_jump->getDest(); 20. ADDRESS jmptbl = ((Const*)pDest->getSubExp2())->getInt(); 21. BasicBlock* pBB = pCfg->newBB(BB_rtls, NWAY, 0); 22. for (int i = 0; ; i++) { 23. ADDRESS uDest = pBF->readNative4(jmptbl + i * 4); 24. if(uDest>=pBF->textLow() && uDest<=pBF->textHigh()){ 25. targetQueue.visit(pCfg, uDest, pBB); 26. pCfg->addOutEdge(pBB, uDest, true); 27. }else{break;} 28. } 29. pBB->updateType(NWAY, i); // 结束::构造多分支控制流结构代码片段 </pre>	<pre> //15.当前过程体的控制流图 //16.存储文件信息, 如跳转表 //17.存储当前基本块的 RTL //18.构建 CFG 辅助数据结构 //19.获取跳转表达式 //20.获取跳转表的地址 jmptbl //21.构造一个 NWAY 基本块 /*22~28.从跳转表中读取地址 uDest, 如果 uDest 在程序范围 内, 则加入到 targetQueue 中; 同时, 增加边<pBB, uDest>到 控制流图 pCfg 中; 否则, 退出 循环*/ //29.设置 pBB 的分支数目 </pre>

算法 4-1 ppc 中基于跳转表的 switch 结构重构的代码片段

算法 4-1 包括两部分，第一部分是 bctr 指令的解码，第二部分是多分支控制流图的构建。图中黑体部分表示程序中使用到的重要的类，大部分都出现在图 4-3 的类图中。其中 Const 类，Location 类和 Binary 类都派生自 Exp 类，分别用于建模常量，内存和二元运算；CaseStatement 类派生自 Statement 类，用于建模多分支语句；DecodeResult 类用来记录一条指令的解码结果。

解码 `bctr` 指令的过程中，首先构造一个多分支语句并设置其跳转表达式，构造 `SWITCH_INFO` 结构体，作为多分支语句的分支信息。最后，将多分支语句加入到解码结果中并设置指令的长度。当 `CFG Builder` 分析到多分支语句时，会进入算法 4-1 中的第 19~29 行的代码，其主要工作是获取多分支语句的跳转表达式，并从跳转表达式中分析出跳转表地址，同时在该多分支语句处构造一个多分支基本块。然后，遍历跳转表中的每一个地址，将其加入到 `targetQueue` 中，用于后续的分析。同时，在相应地址处建立基本块，并与多分支基本块相连，组成多分支控制流子图。

```

00. void Cfg::formalNWay(){
01.     for(std::list<BasicBlock*>::iterator It = m_listBB.begin(); It != m_listBB.end(); It++){
02.         if((*It)->getType() == NWAY){
03.             BasicBlock* switchBB = (*It);
04.             std::vector< BasicBlock*> prevBBs = switchBB->m_InEdges;
05.             if(prevBBs.size() != 1){
06.                 continue;
07.             }
08.             BasicBlock* prevBB = prevBBs.front();
09.             if(prevBB->getType() != TWOWAY){
10.                 continue;
11.             }
12.             BasicBlock* defaultBB ;
13.             for(std::vector< BasicBlock*>::iterator tmpIt = prevBB->m_OutEdges.begin(); tmpIt !=
14.                 prevBB->m_OutEdges.end(); tmpIt++){
15.                 if((*tmpIt) != switchBB){
16.                     defaultBB = (*tmpIt);
17.                 }
18.             }
19.             switchBB->m_InEdges.erase(switchBB->m_InEdges.begin());
20.             switchBB->m_iNumInEdges--;
21.             for(std::vector< BasicBlock*>::iterator tpIt = prevBB->m_InEdges.begin(); tpIt !=
22.                 prevBB->m_InEdges.end(); tpIt++){
23.                 switchBB->m_InEdges.push_back(*tpIt);
24.                 switchBB->m_iNumInEdges++;
25.                 (*tpIt)->m_OutEdges.push_back(switchBB);
26.                 (*tpIt)->m_iNumOutEdges++;
27.                 for(std::vector<PBB>::iterator tmpIt = ((*tpIt)->m_OutEdges.begin()); tmpIt !=
28.                     ((*tpIt)->m_OutEdges.end()); tmpIt++){
29.                     if((*tmpIt) == prevBB){
30.                         (*tpIt)->m_OutEdges.erase(tmpIt);
31.                         (*tpIt)->m_iNumOutEdges--;
32.                         break;
33.                     }
34.                 }
35.             }
36.             for(std::vector< BasicBlock*>::iterator tpIt = defaultBB->m_InEdges.begin(); tpIt !=
37.                 defaultBB->m_InEdges.end(); tpIt++){
38.                 if((*tpIt) == prevBB){
39.                     tpIt = defaultBB->m_InEdges.erase(tpIt);
40.                     defaultBB->m_iNumInEdges--;
41.                 }
42.                 if(tpIt == defaultBB->m_InEdges.end())
43.                     break;
44.             }

```

```

45.     switchBB->m_OutEdges.push_back(defaultBB);
46.     switchBB->m_iNumOutEdges++;
47.     defaultBB->m_InEdges.push_back(switchBB);
48.     defaultBB->m_iNumInEdges++;
49.     removeBB(prevBB);
50.     m_mapBB.erase(m_mapBB.find(prevBB->getLowAddr()));
51. }
52. }
53. }

```

算法 4-2 switch 结构的规整

以下的解说以图 3-3 左图中的控制图中的基本块为例。算法 4-2 的第 03~11 行用来确定 nwBB 的唯一前驱 pBB，如果 nwBB 的前驱不唯一，那么该多分支结构将不被规整。第 12~18 行用来获取 pBB 的另一个出边所指向的基本块 dBB。第 19~20 行删除边 <pBB, nwBB>。第 21~35 行删除与 pBB 相关的边集，包括 pBB 的入边，pBB 的出边等。第 36~44 行将 pBB 从 dBB 的入边链表中删除。第 45~48 行将 dBB 的前驱设置为 nwBB。第 49~50 行将基本块 pBB 从控制流图中删除。

4.5 Goto 语句消除实现

Goto 语句消除主要在图 4-1 中的控制流分析之前完成。然后，经过控制流分析，给程序中的每一个基本块进行分类，包括被复制的基本块。最后，通过代码生成器输出相应的代码。本文的 goto 语句采用迭代式的不动点算法，因为在实际中会出现如图 4-5 的子图(a)所示的控制结构，该结构不能通过 3.3.2 节的四个判断条件一次性检出。

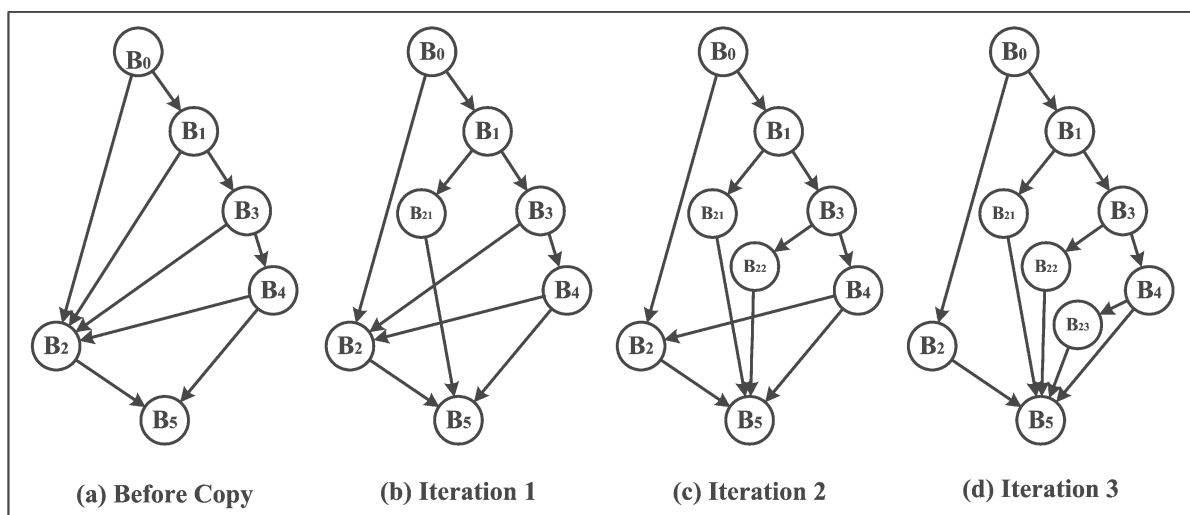


图 4-5 迭代式代码复制消除 goto 语句示意

图 4-5 的子图(a)，按照 3.3.2 节中的四个条件判断，每次只能将基本块 B2 复制一次。在第一次迭代时，将基本块 B2 复制一次，得到 B21，同时修改 B2 的入边，B1 的出边，以及构造新基本块 B21 的出边和入边。其中最重要的一步是将 B2 中的指令都要复制一份到基本块 B21 上，这样才能真正的完成代码复制。同样，在随后的第二次迭代和第三次迭代中，分别完成 B22 和 B23 的复制，以及相应的出边和入边的构造。三

次迭代完成后, 结果如图 4-5 的子图(d)所示。算法 4-3 展示了基本块复制的相关代码。

```

00. std::vector<BasicBlock*>::iterator tmpIt;           //基本块向量迭代器
01. int tmpIdx = 0;
02. std::list<RTL*>::iterator rtlIt;                   //RTL链表迭代器
03. std::list<RTL*> *newRTL = new std::list<RTL*>;      //构造新RTL
04. for(= B2->m_pRtls->begin(); rtlIt != B2->m_pRtls->end(); rtlIt++){
05.     RTL* nRTL = (*rtlIt)->clone();                 //克隆RTL
06.     nRTL->setAddress(uNativeAddr + 4*(tmpIdx++));   //修改RTL的地址
07.     newRTL->push_back(nRTL);                       //将nRTL插入到newRTL链表中
08. }
09. //复制基本块B2, 得B2x, 并修改<B1,B2>为<B1,B2x>
10. BasicBlock* B2x = new BasicBlock(newRTL,B2->getType(),B2->getNumOutEdges());
11. //将新基本块B2x加入到m_listBB和m_mapBB中
12. m_listBB.push_back(B2x);
13. m_mapBB[uNativeAddr] = B2x;
14. //将B1的出边B2删除
15. for(tmpIt = B1->m_OutEdges.begin(); tmpIt != B1->m_OutEdges.end(); tmpIt++){
16.     BasicBlock* tmp = (*tmpIt);
17.     if(tmp == B2){
18.         tmpIt = B1->m_OutEdges.erase(tmpIt);
19.         B1->m_iNumOutEdges --;
20.         if(tmpIt == B1->m_OutEdges.end())
21.             break;
22.     }
23. }
24. //将B2的入边B1删除
25. for(tmpIt = B2->m_InEdges.begin(); tmpIt != B2->m_InEdges.end(); tmpIt++){
26.     BasicBlock* tmp = (*tmpIt);
27.     if(tmp == B1){
28.         tmpIt = B2->m_InEdges.erase(tmpIt);
29.         B2->m_iNumInEdges --;
30.         if(tmpIt == B2->m_InEdges.end())
31.             break;
32.     }
33. }
34. //修改B2x的入边为<B1, B2x >
35. B1->m_OutEdges.push_back(B2x);
36. B1->m_iNumOutEdges ++;
37. B2x ->m_InEdges.push_back(B1);
38. B2x ->m_iNumInEdges ++;
39. //将B2的出边赋给B2x
40. B2x ->m_OutEdges.clear();
41. for(tmpIt = B2->m_OutEdges.begin(); tmpIt != B2->m_OutEdges.end(); tmpIt++){
42.     BasicBlock*tmpOut = (*tmpIt);
43.     //构造B2x的出边
44.     B2x ->m_OutEdges.push_back(tmpOut);
45.     tmpOut->addInEdge(B2x);
46. }

```

算法 4-3 基本块复制相关代码片段

算法 4-3 中的第 00~03 行列举出了整个复制过程中重要的数据结构和迭代器。第 04~08 行遍历被复制基本块 B2 中的 RTL 链表, 并将其克隆增加到新的 RTL 链表

newRTL 中。第 10 行根据基本块 B2 的类型，新的 RTL 链表 newRTL 和基本块 B2 的出边构造一个新的基本块 B2x。第 12~13 行将新复制的基本块加入到基本块链表 m_listBB 和基本块词典 m_mapBB 中，达到真正修改控制流图的目的。第 15~33 行将从 B1 到 B2 的有向边删除。因为控制流图中的边是双向的，因此第 15~23 行将基本块 B2 从基本块 B1 的出边链表中删除，第 25~33 行将基本块 B1 从基本块 B2 的入边链表中删除。第 34~38 行增加有向边<B1, B2x>。第 40~46 行将基本块 B2 的所有出边都赋给基本块 B2x。

4.6 本章小结

本章介绍了 ASMBoom 反编译器的系统框架，实现方案以及智能解码器中最重要的两部分，固有函数消除和指令习语的挖掘。

ASMBoom 反编译器实现的基础是 Boomerang 开源软件，Boomerang 在指令语义描述和可扩展前端设计是 ASMBoom 反编译器中最重要的设计来源。同时，ASMBoom 反编译器在采用了 Boomerang 反编译器的程序组织方式以及程序的数据流分析，类型分析和控制流分析等优化算法。

在固有函数消除方面，首先，介绍了固有函数模板库的构造方法，完整和精确的模板库设计是后续固有函数识别的重要基础。接着，对开源库 vflib 的软件框架及其在 ASMBoom 反编译器中的使用情况进行说明。将图同构或图嵌入等算法应用到反编译领域是一个值得研究的问题。

4.3 节对指令习语的挖掘做了探索性的研究，主要通过挖掘频繁顺序指令集的方式获得可执行程序中的指令习语，并构造指令习语数据库。指令习语的正确翻译对于提高反编译结果的可读性至关重要，在实际工作中应当给予重视。

4.4 节和 4.5 分别介绍了 switch 结构的优化算法实现和 goto 语句消除的实现方法。着重突出图 4-3 中的各个类在反编译中的应用及其使用方法。

总之，ASMBoom 是一款可以灵活扩展，并可以应用到不同领域的反编译软件。例如 4.2 节和 4.3 节中关于固有函数模板库的构建以及指令习语数据库的构建都可以通过 ASMBoom 反编译器辅助实现。

5 实验测试及分析

本文通过整体性能测试，程序结构优化测试以及部分反编译结果示意三方面来评价 ASMBoom 反编译器的性能以及相应的程序结构优化算法的效果。在对比实验中，由于 Hex-Rays 反编译器没有 ppc 程序的反编译功能，因此将 ppc 程序的反编译结果与 RD 反编译器的结果进行对比；将 x86 程序的反编译结果与 Hex-Rays, RD 和 REC 等反编译器的结果进行对比。

5.1 实验方法

本文采用对比的实验方法对 ASMBoom 反编译器的整体性能，程序结构优化算法等展开测试。测试程序，对比平台以及实验平台分别介绍如下。

测试程序分别取自 SPEC CPU 2006 标准测试集和 Olden 标准测试集。来自 SPEC CPU 2006 标准测试集的测试程序分别是 bzip2, h264ref, lbm, mcf 和 sjeng。来自 Olden 标准测试集的测试程序分别是 bisort, health, mst, perimeter, power, treeadd 和 voronoi。

另外，还有一些测试集来自 Github 等网站，分别是处理 XML 文件的 xml 测试集^[93]，实现^[94]中描述的 AWK 语言的 awk 编译器^[95]，基于正则表达式的文本搜索工具 grep^[96]以及开源数据库软件 sqlite^[97]。来自 Github 的程序既可以用来评测 x86 汇编程序的反编译效果，还可以用来对比基于子图同构的固有函数检测算法的效果。

对比平台分别是工业界的 Hex-Rays 反编译器，学术界流行的 RD 反编译器和时间比较长久的 REC 反编译器。Hex-Rays 反编译器是 IDA 反汇编工具的一个插件，是工业界被广泛使用的反编译器，本文所使用的 Hex-Rays 的版本是 6.1.110315(32 bit)。REC 是一款交互式的反编译器，支持多种文件格式和指令集。REC 并非开源软件，但是可以免费下载使用。RD 提供了一个网页界面，可以免费在线使用，并导出实验结果。

实验平台的基本配置如下：Intel Core Quad 3.10GHz CPU，8GB 内存，Windows 7 操作系统。实验用软件是在 Boomerang 软件基础上扩展的，以汇编作为输入，支持 x86 和 ppc 两种指令系统的 ASMBoom 反编译软件。

5.2 反编译器整体性能测试

为了测试 ASMBoom 反编译 x86 汇编程序的整体性能，使用 MSVC 2010 编译器，在-O2 优化选项下，分别将 xml 测试集，awk 测试集，grep 测试集和 sqlite 测试集编译成可执行程序。然后，使用 IDA 对可执行程序分别进行反汇编处理，得到每一个测试集的汇编代码，将这些汇编代码作为 ASMBoom 反编译器的输入，最终得到所有测试集类 C 代码输出。而 Hex-Rays 反编译器和 RD 反编译器直接以可执行程序作为输入，经过反编译处理之后生成相应的类 C 代码。

因为反编译结果最重要的功能是代码理解，而影响代码理解的最重要的两个因素

是代码的长度以及代码中的 goto 语句的数目。所以，在对比评价中，着重突出这两项指标。表 5-1 中的反编译行一列表示反编译器输出的 C 代码行数，goto 数/占比一列表示每种反编译器输出的 C 代码中 goto 语句的数目以及大致多少行有一个 goto 语句。RD 反编译器在反编译 sqlite 可执行程序中，不论 decompilation settings 如何设置，都会在反编译过程中出错，因此其统计数据可能不是非常完整。

表 5-1 x86 反编译结果对比

测试集	Hex-Rays		RD		ASMBoom	
	反编译行	goto 数/占比	反编译行	goto 数/占比	反编译行	goto 数/占比
xml	1033	3/344	2170	32/68	656	16/41
awk	12512	282/44	43444	769/56	9823	368/27
grep	11513	275/42	30763	631/49	6373	366/17
sqlite	136340	2874/47	--	--	115692	6410/18
Total	161398	3434/47	76377	1432/53	132544	7160/18

Goto 占比的定义为反编译行数与 goto 语句数目的比值。在表 5-1 中，从反编译行来看，四个测试集的复杂程序从高到低排序为：sqlite，awk，grep 和 xml。Goto 语句的数目在三个反编译器中显示出与测试集复杂度相一致的增长趋势，即程序复杂度越高，goto 语句也越多。虽然这不是一个通理，但是从逻辑的角度来看，程序的复杂度通常体现在其控制结构的复杂度上。而控制结构越复杂，越不规整，对于反编译中的结构化算法的分析难度也越大，结构化算法处理不了的结构就会用 goto 结构来替代。

另一方面，RD 生成的代码应当是最多的，且 goto 语句也是最多的。而 ASMBoom 生成的代码最少，但是 goto 语句远远要多于 Hex-Rays。在 goto 占比的比较中，ASMBoom 的占比最低，即 goto 语句的密度比较高。而 Hex-Rays 的 goto 占比次之，最高的是 RD 反编译器。可见，ASMBoom 在反编译优化算法方面，还存在许多可以提高的空间。

在测试 ASMBoom 反编译器反编译 ppc 汇编程序的整体性能时，首先利用 powerpc-linux-gcc-4.4 交叉编译器，采用 -O0 优化，将表 5-2 所示的测试程序分别编译成汇编程序和可执行程序。其中的汇编程序用作 ASMBoom 反编译器的输入，而可执行程序用作反编译器 RD 的输入。然后，通过实验分别评价 ASMBoom 反编译器的整体性能以及对应的结构优化技术对生成代码在结构上的改善，并与学术界非常流行的 RD 反编译器结果进行对比。

表 5-2 展示的是 ASMBoom 反编译器和 RD 反编译器在 ppc 程序上的反编译结果的对比情况。从反编译行来看，ASMBoom 最终生成的代码量要少于 RD 反编译器的代码量，ASMBoom 生成代码中的 goto 语句数目也要少于 RD 反编译器。但是，在 goto 占比评价指标来看，RD 的高 goto 占比显示出其低 goto 密度。但是，从实际考察来看，应该还是 ASMBoom 反编译器较 RD 反编译器有更大的优势。另外，对于 Olden 测试集中的测试程序，其生成的代码量少且 goto 语句较少。说明两个编译器各有所长。

从 ASMBoom 反编译器对 x86 和 ppc 程序的反编译效果来看，在生成代码量方面，ASMBoom 具有很大的优势。在 goto 语句数目上也要优于 RD 反编译器，而劣于

Hex-Rays。在 goto 占比上，都要差于 Hex-Rays 和 RD 反编译器。

表 5-2 ppc 程序反编译结果对比

测试集	RD			ASMBoom		
	反编译行	goto 数目	goto 占比	反编译行	goto 数目	goto 占比
bisort	300	0	∞	399	4	99
health	665	0	∞	771	3	257
mst	453	0	∞	669	4	167
perimeter	700	0	∞	791	9	88
power	282	0	∞	877	5	175
treeadd	87	0	∞	125	0	∞
voronoi	1061	2	531	2137	9	237
bzip2	24534	300	82	19597	531	37
h264ref	238752	2141	112	122864	1898	65
lbm	571	8	71	2166	22	98
mcf	2360	48	49	2424	57	43
sjeng	86668	895	97	27251	705	39
Total	356433	3394	105	180071	3247	55

注： ∞ 表示程序中没有 goto 语句。

5.3 程序结构优化测试

本节主要测试 ASMBoom 反编译器在 switch 结构解码和 goto 语句消除方面所做的进展，并与 Boomerang 反编译器进行对比，突出 ASMBoom 反编译器结构优化算法相对 Boomerang 反编译器的改进和提高。另外，在固有函数消除方面，通过对比实验说明基于子图同构算法的内联固有函数的消除算法较其它算法的优势。

表 5-3 ASMBoom 反编译器与 Boomerang 反编译器在结构优化方面的对比

测试集	过程数	源码行	汇编行	反编译行对比	switch 对比	goto 对比	goto 占比对比
bisort	13	350	905	399/399	0/0	4/4	99/99
health	17	504	1921	771/771	0/0	3/3	257/257
mst	16	428	1207	669/669	0/0	4/4	167/167
perimeter	12	484	1157	791/775	0/0	9/20	88/39
power	17	622	2257	877/877	0/0	5/5	175/175
treeadd	4	245	251	125/125	0/0	0/0	∞/∞
voronoi	45	1151	4302	2137/2137	0/0	9/9	237/237
bzip2	122	8293	35995	19597/18943	4/4	531/552	37/34
h264ref	593	51578	249791	122864/121183	12/12	1898/2255	65/53
lbm	22	1155	8076	2166/2166	0/0	22/22	98/98
mcf	24	2685	4743	2424/2414	0/0	57/61	43/40
sjeng	144	13847	53617	27251/23226	17/17	705/721	39/32
Total	1029	81342	364222	180071/173685	33/33	3247/3656	55/48

表 5-3 中的测试集采用 powerpc-linux-gcc-4.4 交叉编译器的-O0 优化，编译成汇编

程序，分别作为 Boomerang 和 ASMBoom 反编译器的输入。表 5-3 中的过程数一行代表测试程序源码中的过程体数目，源码行一行表示测试程序中的源码行数，汇编行一行表示经过 powerpc-linux-gcc-4.4 交叉编译器的-O0 优化之后得到的汇编代码的行数，反编译行对比一行对比 ASMBoom 反编译结果的行数（"/"前）与 Boomerang 反编译结果的行数（"/"后），switch 对比一行对比源码中的 switch 数目（"/"前）与 ASMBoom 反编译结果中的 switch 数目（"/"后），goto 对比一行对比 ASMBoom 反编译结果中的 goto 数目（"/"前）与 Boomerang 反编译结果中的 goto 数目（"/"后），goto 占比对比一行对比 ASMBoom 反编译结果中的 goto 占比（"/"前）与 Boomerang 反编译结果中的 goto 占比（"/"后）。

通过分析反编译行对比一栏可以发现，只要有 goto 语句删除，就会有代码数量的上升，因为本文中的 goto 语句删除算法的依据是 3.3.2 节中描述的基于代码复制的 goto 语句消除策略，且结合 goto 占比的变化情况，发现平均每次代码复制要复制约 7 行的代码。进行 goto 语句删除之后，goto 占比都有所上升，说明代码中的 goto 语句的密度在下降，对于提高反编译结果的可读性有一定作用。

在对比各种反编译器在固有函数消除方面的工作时，采用和表 5-1 相同的测试集编译方法。然后，将可执行程序分别交给 Hex-Rays，REC 和 RD 作为输入，而将可执行程序经 IDA 反汇编的结果作为 ASMBoom 反编译器的输入。结果对比如表 5-4。

表 5-4 固有函数消减对比

测试集	固有函数	调用次数	检出次数			
			Hex-Rays	REC	RD	ASMBoom
xml	strlen	7	3	0	0	6
	strcpy	9	0	0	0	9
	memset	0	3	3	3	--
	strcmp	0	0	0	0	0
	memcmp	0	0	0	0	0
awk	strlen	36	11	0	0	38
	strcpy	7	0	0	0	18
	memset	1	6	19	10	--
	strcmp	17	16	0	0	11
	memcmp	0	0	0	0	0
grep	strlen	35	11	0	0	22
	strcpy	17	0	0	0	9
	memset	3	7	11	9	--
	strcmp	36	11	0	0	5
	memcmp	1	0	0	0	1
sqlite	strlen	68	3	0	--	4
	strcpy	0	0	0	--	11
	memset	255	203	195	--	--
	strcmp	104	72	0	--	70
	memcmp	57	0	0	--	11

表 5-4 中的固有函数的详细说明见表 4-3。表 5-4 中的'--'代表相应数据不可得。ASMBoom 没有构建 `memset` 的模板，因为 `memset` 在程序中表现的非常灵活，模板种类太多。首先，可以肯定的是 REC 和 RD 在检测表 4-3 所列举的内联固有函数时不是非常到位，只在 `memset` 固有函数的检测上比较突出，并且在许多情况下还要好于 Hex-Rays。而 Hex-Rays 只有 `strcmp` 固有函数的检测比 ASMBoom 所采用的方法突出，其余都要劣于 ASMBoom。可见，基于子图同构的内联固有函数的检测技术在结果上要优于其它反编译器的方法，虽然会增加反编译时间，但是这样的增加是有意义的。

但是，不是所有的反编译器都能够完全检测可执行代码中内联的固有函数，如图 3-8 所示的 Hex-Rays 的结果。这也从侧面反映了内联固有函数的识别是一个非常具有挑战性的工作。基于图拓扑嵌入的方法，或许是未来解决该问题的思路之一。

5.4 反编译结果实例

本节着重对比突出第 3 章中的 `goto` 语句消除算法，`switch` 结构规整两方面的结构优化结果，并且将这些结果与主流反编译器 RD 和 Hex-Rays 进行对比。

<pre>signed int __cdecl main(int a1, int a2){ __int32 v2; // esi@1 signed int result; // eax@2 v2 = strtol(*(const char **)(a2 + 4), 0, 10); if (strtol(*(const char **)(a2 + 8), 0, 10) <= 3 (result = 3, v2 <= 3)) result = 5; return result; }</pre>	<pre>int main(int argc, char* argv[]){ int a = atoi(argv[1]); int b = atoi(argv[2]); int c; if((a>3) && (b>3)) c = 3; else c = 5; return c; }</pre>
(a) <i>Hex-Rays</i>	(b) <i>Source Code</i>
<pre>int main(int argc, char ** argv) { int32_t v1 = (int32_t)argv; strtol((int8_t *)*(int32_t *) (v1 + 4), NULL, 10); strtol((int8_t *)*(int32_t *) (v1 + 8), NULL, 10); return 5; }</pre>	<pre>void Func_2714(){ Func_2710(); Func_2710(); if ((R30 <= 0x3)){ R3 = 0x5; } else { R3 = 0x3; if ((R3 <= 0x3)){ goto L1; } } return ; } void Func_2714(){ Func_2710(); Func_2710(); if ((R30 <= 0x3)){ R3 = 0x5; } else { R3 = 0x3; if ((R3 <= 0x3)){ R3 = 0x5; } } return ; }</pre>
(c) <i>RD (Retargetable Decompiler)</i>	(d) <i>ASMBoom_org</i> (e) <i>ASMBoom</i>

图 5-1 各个反编译器输出对比

图 5-1 的子图 (b) 是一个简单的具有复合条件表达式的 C 程序例子。子图 (b)

中的 C 程序经过 gcc 4.8 编译器在 -O2 优化选项编译得到 x86 可执行程序，该可执行程序在 Hex-Rays 反编译处理之后得到如子图 (a) 所示的结果。子图 (b) 中的 C 程序经过 powerpc-linux-gcc-4.4 交叉编译器的 -O2 优化编译得到 ppc 可执行程序，该可执行程序经过 RD 反编译器处理之后得到如子图 (c) 所示的结果。子图 (b) 中的 C 程序经过 powerpc-linux-gcc-4.4 交叉编译器的 -O2 优化编译得到 ppc 汇编程序，汇编程序经过一个过滤器，过滤掉了程序中的函数调用名之后作为 ASMBoom 反编译器的输入。该汇编程序经过未增加代码复制消除 goto 语句算法的 ASMBoom 反编译器处理之后，得到如子图 (d) 所示的结果。汇编程序经过增加了代码复制消除 goto 语句算法的 ASMBoom 反编译器处理之后，得到如子图 (e) 所示的结果。

对比发现，RD 反编译器的反编译结果出错，Hex-Rays 反编译器的结果虽然正确，但是逻辑复杂，ASMBoom 反编译器在增加了基于代码复制消除 goto 语句算法之后，确实消除了 goto 语句。总体来讲，派生自 Boomerang 反编译器的 ASMBoom 反编译器的反编译结果的抽象程度不是非常高，可以看做是具有高级结构的汇编程序，而 Hex-Rays 的抽象程度比较高，RD 在 ppc 程序翻译上还不是非常成熟。

图 5-2 最左边展示的是图 5-1 中的子图 (b) 的 C 程序经过 powerpc-linux-gcc-4.4 交叉编译器的 -O2 优化编译得到 ppc 汇编程序。图 5-2 中间展示的是未经过代码复制之前的控制流图，显然在该控制流图中存在着一类类似于图 3-7 中的控制流子图所示的交错边。因此，在经过代码复制之后，得到图 5-2 最右边的控制流图。

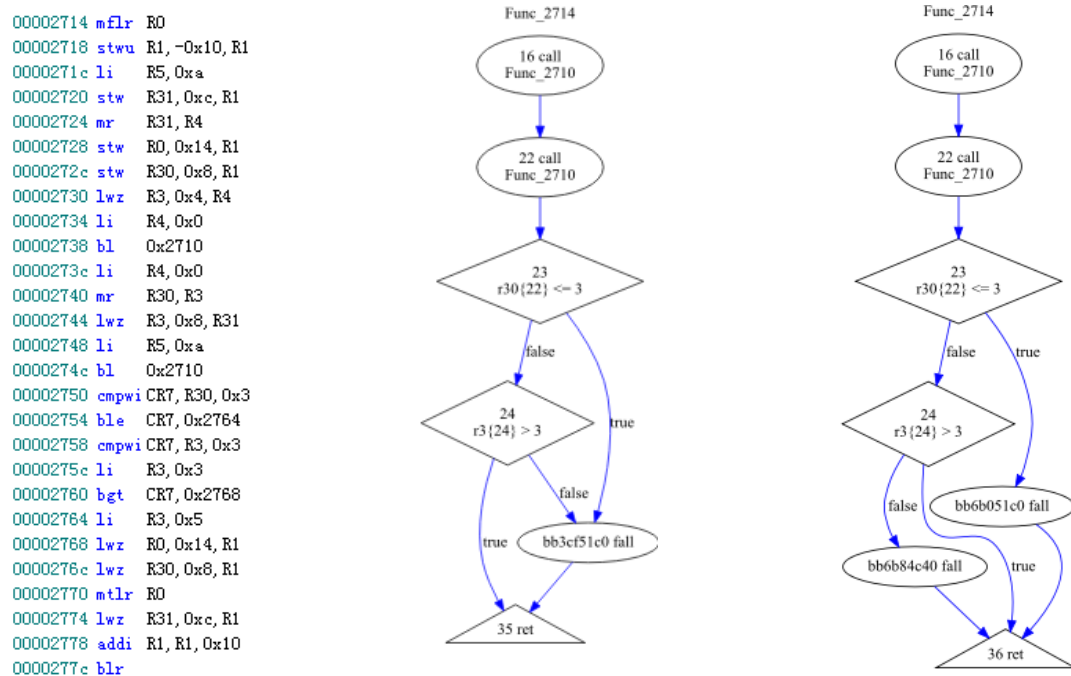


图 5-2 代码复制截图示意

图 5-2 中间的控制流图经过反编译后续处理之后，得到图 5-1 的子图 (d) 中的类 C 代码。图 5-2 最右边的控制流图经过反编译后续处理之后，得到图 5-1 的子图 (e) 中的类 C 代码。

在 ppc 可执行程序的 switch 结构处理方面，分别用 gcc 4.8 的 -O0 优化选项下编译图 5-3 中子图 (a) 中的程序，最终被翻译成跳转表形式的 switch 结构，得到 switchx86 可执行程序。将 switchx86 可执行程序用 Hex-Rays 反编译，结果中并没有构造出 switch 结构，而是产生如图 5-3 中子图 (c) 所示的输出。

<pre> void main(int argc, char* argv[]){ int a = atoi(argv[1]); int b = atoi(argv[2]); int c = 0; int ch = argc; switch(ch){ case 1: c = a + b; break; case 2: c = a - b; break; case 3: c = a * b; break; case 4: c = a / b; break; case 5: c = a + b * a; break; } return c; } </pre>	<pre> R0 = R9; switch(R0){ case 0x1: R9 = *(R31 + 0x14); R0 = *(R31 + 0x10); *(unsigned int*)(R31 + 0xc) = R9 + R0; break; case 0x2: R9 = *(R31 + 0x14); R0 = *(R31 + 0x10); *(int*)(R31 + 0xc) = R9 - R0; break; case 0x3: R9 = *(R31 + 0x14); R0 = *(R31 + 0x10); *(unsigned int*)(R31 + 0xc) = R9 * R0; break; case 0x4: R9 = *(R31 + 0x14); R0 = *(R31 + 0x10); *(unsigned int*)(R31 + 0xc) = R9 / R0; case 0x5: R0 = *(R31 + 0x10); R0 = *(R31 + 0x14); *(unsigned int*)(R31 + 0xc) = (R0 + 0x1) * R0; break; default: break; } </pre>
(a) <i>Source Code</i>	(b) <i>ASMBoom</i>
<pre> unsigned int main(unsigned int a1, int a2){ unsigned int result; // eax@1 atoi(*(const char **)(a2 + 4)); atoi(*(const char **)(a2 + 8)); result = a1; if (a1 <= 5) JUMPOUT(__CS__, off_8048558[a1]); return result; } </pre>	<pre> int main(int argc, char ** argv) { // 0x100004d4 atoi(*argv); return atoi(*argv); } </pre>
(c) <i>Hex-Rays</i>	(d) <i>RD (Retargetable Decompiler)</i>

图 5-3 switch 结构反编译对比

将图 5-3 中子图 (a) 的 C 程序用 powerpc-linux-gcc-4.4 交叉编译器的-O0 优化编译得到 ppc 汇编程序, 并用 ASMBoom 反编译得到图 5-3 中子图 (b) 的结果, 为了展示简洁, 省略了 switch 结构前后的代码段。将图 5-3 中子图 (a) 的 C 程序用 powerpc-linux-gcc-4.4 交叉编译器的-O0 优化编译得到 ppc 可执行程序, 最终经过 RD 反编译器处理得到如图 5-3 中子图 (d) 所示的结果。

对比发现, ASMBoom 将基于跳转表的 switch 结构恢复地比较完整, 并且对表达式进行了优化组合, 如将原来 case 5 中的 $a+b*a$ 翻译成了 $(b+0x1)*a$ 的形式。另外, 由于汇编程序是-O0 优化的, 因此反编译结果比较繁琐, 每一次从内存取值的过程都被翻译出来。Hex-Rays 没有反编译出 switch 结构, 说明其没有很好的利用 IDA 的反汇编输出, 因为根据 3.2.1 节的分析, IDA 已经分析出了 switch 的跳转表, 只是没有进一步分析, 将控制流程图构造完整。图 5-4 展示了基于跳转表翻译的 switch 结构被 ASMBoom 翻译以及规整之后的结果。

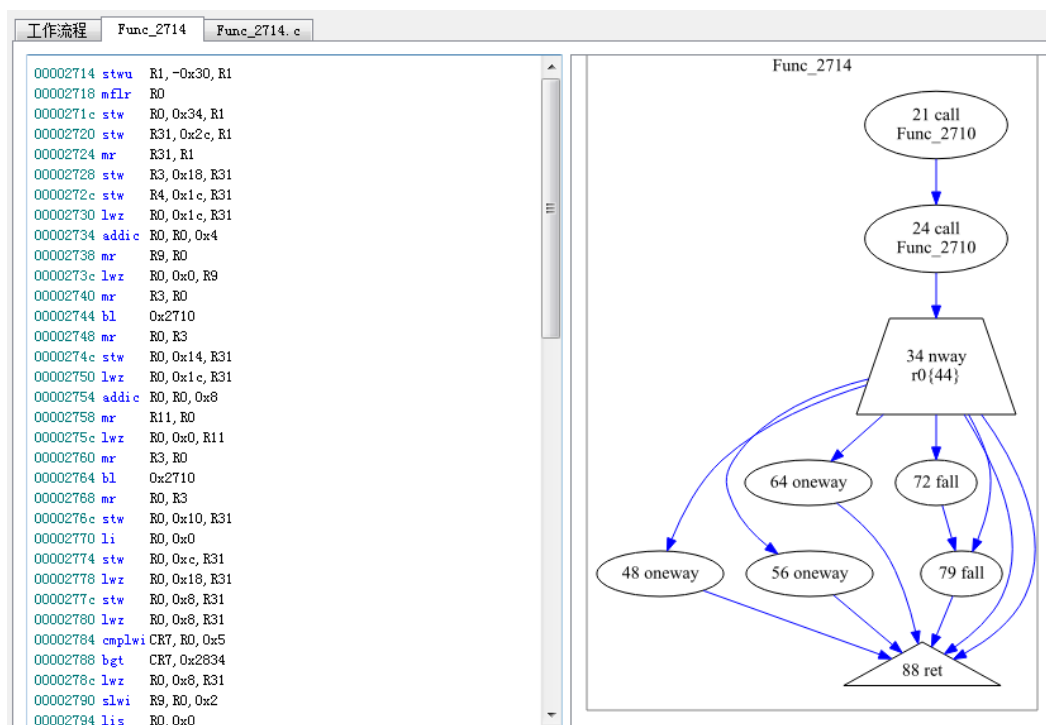


图 5-4 ASMBoom 反编译器对基于跳转表的 switch 结构的处理及规整

图 5-4 中的 34 号基本块是一个多分支结点, 分别有跳往编号为 48,64,56,72,79 和 88 的基本块的边。边<34, 88>模拟的是 default 分支边, 而边<72, 79>则是由于在源码中的 case 4 没有 break 语句而引入的分支。另外, 由于仍然可能出现 case 为 5 的可能, 因此边<34, 79>被保留。图 5-4 嵌套在 34 号基本块外面的分支结构被删除并规整。

基于二叉树的 switch 结构的恢复, 图 5-5 是基于二叉树的 switch 结构的规整对比。左边的子图(a)是通过 ASMBoom 构建的原始的程序控制流程图; 右上方的子图(b)是经过算法 3-3 规整后得到的控制流程图, 算法将二叉树结构转换成了一个多分支结构; 右下方的子图(c)是 RD 反编译器规整二叉树得到的结果。对比(b)和(c)发现, RD 反编译器

在规整方面做得并不是非常到位，整个分支结构被规整成为一个 if-else 语句的两个分支中分别嵌套的两个 switch 语句组成。本质原因就在于 RD 对于 NONTERMINAL 结点的处理不是非常完全。

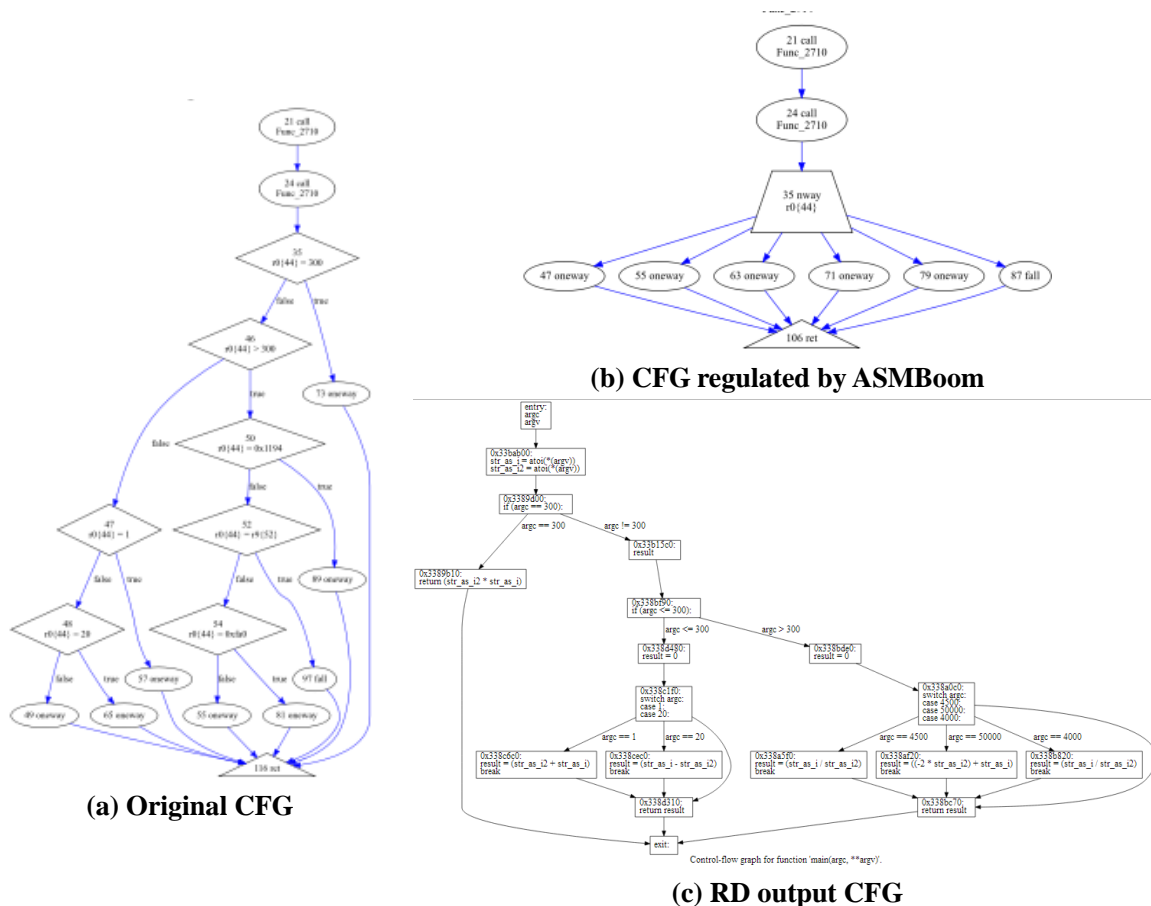


图 5-5 基于二叉树的 switch 程序规整对比

5.5 本章小结

本章以 SPEC CPU 2006 标准测试集, Olden 标准测试集以及从 Github 上下载的 xml 测试集, awk 测试集, grep 测试集和 sqlite 测试集为测试用例, 分别就 ASMBloom 反编译器的整体性能以及程序结构优化算法等展开测试。在测试的过程中, 为了进一步说明反编译器和相应算法的有效性, 分别和 Hex-Rays, RD, Boomerang 和 REC 等反编译器的反编译结果和算法性能进行了对比。实验结果表明, ASMBloom 反编译器在某些方面是优于其它反编译器的。

在本章的最后一节, 给出了 ASMBloom 反编译器的部分反编译结果并与主流反编译器的结果进行对比说明。

6 结论与展望

6.1 结论

随着智能设备以及互联网与物联网的普及，程序的安全性问题越来越突出。反编译作为代码监管中一项重要的技术，对于理解代码至关重要，而程序的结构优化作为反编译研究中的重要内容，可以提高代码的可读性和逻辑性。

因为反编译的输入通常是编译优化后的结果，因此要在反编译过程中优化程序的结构必须了解编译对程序结构的优化。编译对于程序的结构优化主要体现在分支结构的优化，循环结构的优化，函数内联等三方面。尤其是随着体系结构的升级，循环的向量化和并行化优化更加深入。另外，一些软件保护技术通过迷乱程序的控制流来达到阻止逆向分析的目的。所有这些优化和防护措施都无形中增加了反编译中的结构分析和优化的复杂度。

反编译器在发展过程中，其架构，中间语言，开发方式和反编译的思路都在不断地发生着变化。基于这些特点，本文提出了自己的反编译器框架，特点在于提高 IR 的抽象程度，通用性和兼容性以及提高反编译器的扩展性。最终设计并开发 ASMBoom 反编译器。

以 ASMBoom 反编译器为基础，在程序的控制流结构上实现不同形式的 switch 结构的解码和规整，基于代码复制消除 goto 语句的技术和基于子图同构的内联固有函数消除算法。并采用 SPEC CPU 2006 标准测试集，Olden 标准测试集和 Github 上的部分开源程序作为测试用例，以 Hex-Rays, RD, Boomerang 和 REC 等主流反编译器作为对比平台，评价 ASMBoom 反编译器的整体性能和部分结构优化算法的效果。

实验结果说明，ASMBoom 反编译器在整体性能上比 Boomerang 软件都有所提高，在某些方面要优于 Hex-Rays 和 RD，如 switch 结构的翻译以及内联固有函数的消除等方面。

本文的主要贡献体现在如下几个方面：

(1) 对比研究问题的思路。将编译和反编译的优化进行对比，将发展最快的编译方向的最新技术应用到反编译领域是非常有意义的。例如基于子图同构的内联固有函数的消除技术的思路就来源于编译器优化中指令习语的识别技术。另外，在各个反编译框架以及各个反编译技术的研究上面，都广泛采用了对比分析的思路。对比分析可以增加对于问题的理解深度。

(2) 利用图论的知识和算法解决反编译领域中的问题。编译器，反编译器和编程语言的设计是与图论息息相关的领域，因为这些领域需要借助图来建模相应领域中的对象以及对象之间的关系。本文的基于二叉树的 switch 结构的优化，指令习语的检测和分析以及基于子图同构的内联固有函数消除等都使用了图论的相关知识和算法。

(3) 综合分析问题。在反编译框架的设计上面，综合了 Boomerang, Phoenix,

BAP 和 RD 等多种反编译器的设计思路。最终设计的反编译可以灵活的扩展指令系统以及文件格式，生成更加抽象的中间代码等。

6.2 进一步工作展望

反编译是一个不断发展的领域，做好反编译研究对于代码安全分析，代码算法理解以及实现软件的定制和扩展都有重要的作用。下一步的研究将从以下几个方面展开。

(1) 扩展中间语言生成模块，例如生成 BIL, REIL 和 LLVM IR 等中间语言，从而可以将 ASMBloom 反编译器的前端分析交给其它逆向分析工具。实现更大领域范围内的对比和研究。

(2) 基于子图同构的内联固有函数检测的最大缺陷就是要求模板控制流图要非常精确，这样才能够在目标控制流图上检测到相应的模板。但是，有时候由于编译优化的作用，程序的控制流图会被修改，尤其是在循环的优化方面，可能出现一些等价而非同构的控制结构。这时候需要利用拓扑嵌入的方法在目标控制流图上检测相应的模板。基于子图同构的内联固有函数检测可以扩展到用户自定义函数的内联检测中，这样可以进一步减少反编译结果的代码，增加程序的抽象度，提高反编译结果的可读性。

(3) 指令习语往往与编译器，编译优化及编译优化次序，目标平台等紧密相关。因此，指令习语往往千变万化。在分析真实程序的时候，往往不知道编译器，编译器优化及编译优化次序和目标平台等信息。因此，构造一个指令习语库，对于提高指令习语分析的精确性将有很大意义。

(4) 为了避开一些程序迷乱技术对于静态逆向分析的影响，应该增加动态分析部分。即首先通过代码注入工具如 PIN^[100], Valgrind^[101]等，动态监测程序的运行，从运行信息上构建程序的控制流图，然后在目标控制流图上进行反编译分析。这样可以很好的避开一部分程序迷乱技术。

(5) 延伸基于搜索的反编译思路，进一步缩小搜索的范围。以后的搜索可能针对的不再是一个过程体。基于代码片段的代码复用其实也非常多见，因此，搜索可能小到一个控制流子图，一个循环，一个基本块，甚至是一个程序切片。

致 谢

卧薪二十载，修身继绝学，回目来时路，恩师点迷津，同窗天下事，父母严慈宽。

感谢我的导师赵银亮教授这五年来的悉心指导和关怀。初识赵老师是在大三的《编译原理》课上，赵老师黑板上的语法树和自动机所散发出来的学术魅力和人格魅力，将我带入编译领域。进入研究生阶段，赵老师在科研上的执着和对学科发展方向的精准把握有如春风化雨，让我受益匪浅。在反编译的研究过程中，赵老师的意见与鼓励让我坚定信念，倍受鼓舞。衷心感谢赵老师的教导和鼓励，人生中得到赵老师教诲，道路也明朗了许多，宽阔了许多。

感谢我的师兄师姐们，他们是曾庆花，李远成，刘斌，李美容，李玉祥，种翔，梁增玉。在刚进入实验室的时候，他们给予我很大的帮助；在日常生活中，亦师亦友，探讨科研，分享人生经验。

感谢我的同门战友武万杰，王启明，何守伟，程方东和高欢，与他们一起学习进步是件非常快乐的事情。感谢我的同学武万杰，师弟张磊和刘凯，在最关键时刻，是他们和我一起攻克难关，完成项目。感谢七年的舍友王强，美好的回忆历历在目。

特别感谢我的父母，一直以来无私的支持和培养，恩情深重，永生难报。感谢我的爷爷，他的苦难经历，他给我讲的所有故事以及他对我的爱让我更快更健康地成长。一并祝愿他们福寿安康。

最后，向一直关心我的亲友师长深表谢意。

参考文献

- [1] Linn, C. and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. in Proceedings of the 10th ACM conference on Computer and communications security. 2003. Washington D.C., USA: ACM.
- [2] Capstone. Lightweight multi-platform, multi-architecture disassembly framework <http://www.capstone-engine.org/index.html>. 2015 March.
- [3] Cifuentes, C. Reverse Compilation Techniques. Compiling (Electronic computers), 1994.
- [4] Brumley, D., et al. BAP: a binary analysis platform. in Computer aided verification. 2011: Springer.
- [5] O Sullivan, P., et al., Retrofitting security in cots software with binary rewriting, in Future Challenges in Security and Privacy for Academia and Industry. 2011, Springer. p. 154-172.
- [6] Křoustek, J., F. Pokorný and D. Kolář, A new approach to instruction-idioms detection in a retargetable decompiler. Computer Science and Information Systems, 2014. 11(4): p. 1337-1359.
- [7] Fracture: Inverting LLVM's Target Independent Code Generator. <https://github.com/draperlaboratory/fracture>. 2015 March.
- [8] Lattner, C.A., Macroscopic data structure analysis and optimization. 2005, University of Illinois at Urbana-Champaign.
- [9] Zakai, A. Emscripten: an LLVM-to-JavaScript compiler. in Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. 2011. Portland, Oregon, USA: ACM.
- [10] Enck, W., et al. A Study of Android Application Security. 2011.
- [11] Security Flaws in Universal Plug and Play Unplug Dont Play. <https://hdm.io/writing/originals/SecurityFlawsUPnP.pdf> . 2015 March.
- [12] Maruyama, K. and T. Omori. A security-aware refactoring tool for Java programs. in Proceedings of the 4th Workshop on Refactoring Tools. 2011. Waikiki, Honolulu, HI, USA: ACM.
- [13] Yang, X., et al., Finding and understanding bugs in C compilers. SIGPLAN Not., 2011. 46(6): p. 283-294.
- [14] Wheeler, D.A. and D.A. Wheeler, Fully Countering Trusting Trust through Diverse Double-Compiling. Computer science, 2010.
- [15] Van Emmerik, M.J., Static single assignment for decompilation. 2007, The University of Queensland.
- [16] Ramsey, N., M.F. Fern and Ndez, Specifying representations of machine instructions. ACM Trans. Program. Lang. Syst., 1997. 19(3): p. 492-524.
- [17] Cifuentes, C. and S. Sendall. Specifying the semantics of machine instructions. in Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on. 1998.
- [18] Fonseca, J., Interactive Decompilation. 2006.
- [19] Kinder, J. and H. Veith. Jakstab: A static analysis platform for binaries. in Computer Aided Verification. 2008: Springer.
- [20] Kroustek, J. and F. Pokorny, Reconstruction of instruction idioms in a retargetable decompiler. Computer Science and Information Systems, Federated Conference on, 2013: p. 1519 - 1526.
- [21] Durfina, L., et al. Detection and Recovery of Functions and their Arguments in a Retargetable Decompiler. in Reverse Engineering (WCRE), 2012 19th Working Conference on. 2012.
- [22] Dagger decompiler. <http://dagger.repzret.org/>. 2015 March.
- [23] mcsema. Translating native to LLVM IR. <https://github.com/trailofbits/mcsema>. 2015 March.

-
- [24] Urfina, L.V.D. and D.K.A.V. R. Generic Detection of the Statically Linked Code. 2013. Spi\{s}sk\{a} Nov\{a} Ves, SK: FEI TU in Ko\{s}ice.
 - [25] Vawtrak 木马. <http://now.avg.com/banking-trojan-vawtrak-harvesting-passwords-worldwide/>. 2015 March.
 - [26] Schwartz, E.J., et al. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. in Proceedings of the USENIX Security Symposium. 2013.
 - [27] Balakrishnan, G. and T. Reps, Analyzing Memory Accesses in x86 Executables. Lecture Notes in Computer Science, 2004: p. 5-23.
 - [28] Lee, J., T. Avgerinos and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. in NDSS. 2011.
 - [29] Khoo, W.M., Decompilation as search. University of Cambridge, Computer Laboratory, Technical Report, 2013(UCAM-CL-TR-844).
 - [30] Guilfanov, I., Decompilers and beyond. Black Hat USA, 2008.
 - [31] Wienskosi, E. Switch Statement Case Reordering FDO. in GCC Summit. 2006: Citeseer.
 - [32] Hwu, W.W. and P.P. Chang, Inline function expansion for compiling c programs. ACM SIGPLAN Notices, 1989. 24(7): p. 246-257.
 - [33] Dunaev, D. and L. Lengyel, Method of Software Obfuscation Using Petri Nets. Central European Conference on Information & Intelligent Systems, 2013.
 - [34] Ge, J., Control flow based obfuscation. In DRM '05: Proceedings of the 5th ACM workshop on Digital rights management, 2005: p. 83 - 92.
 - [35] Yao, X., et al., A Method and Implementation of Control Flow Obfuscation Using SEH. Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on, 2012. 48(11): p. 336 - 339.
 - [36] Williams, M.H., Generating Structured Flow Diagrams: The Nature of Unstructuredness. Computer Journal, 1977(1): p. 45-50.
 - [37] Erosa, A.M. and L.J. Hendren. Taming control flow: A structured approach to eliminating goto statements. in Computer Languages, 1994., Proceedings of the 1994 International Conference on. 1994: IEEE.
 - [38] Oulsnam, G., Unravelling unstructured programs. The Computer Journal, 1982. 25(3): p. 379-387.
 - [39] Ramshaw, L., Eliminating go to's while preserving program structure. J. ACM, 1988. 35(4): p. 893-920.
 - [40] Lichtblau, U., Decompilation of control structures by means of graph transformations, in Mathematical Foundations of Software Development. 1985, Springer. p. 284-297.
 - [41] Allen, F.E. Control flow analysis. in Proceedings of a symposium on Compiler optimization. 1970. Urbana-Champaign, Illinois: ACM.
 - [42] Engel, F., et al. Enhanced structural analysis for C code reconstruction from IR code. in Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems. 2011. St. Goar, Germany: ACM.
 - [43] Cifuentes, C. Structuring decompiled graphs. in Compiler Construction. 1996: Springer.
 - [44] Johnson, R., D. Pearson and K. Pingali. The program structure tree: Computing control regions in linear time. in ACM SigPlan Notices. 1994: ACM.
 - [45] Wei, T., et al. Structuring 2-way branches in binary executables. in Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International. 2007: IEEE.
 - [46] Wei, T., et al., A new algorithm for identifying loops in decompilation, in Static Analysis. 2007, Springer. p. 170-183.
 - [47] Yakdan, K., et al., No More Gotos: Decompilation Using Pattern-Independent Control-Flow

- Structuring and Semantics-Preserving Transformations. 2015.
- [48] Offner, C.D., Notes on Graph Algorithms Used in Optimizing Compilers. 1995.
 - [49] Authors, U., Approximate Graph Clustering for Program Characterization. *Acm Transactions on Architecture & Code Optimization*, 2012. 8(4): p. 73-94.
 - [50] Park, E., J. Cavazos and M.A. Alvarez. Using graph-based program characterization for predictive modeling. in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012. San Jose, California: ACM.
 - [51] Melnik, S., H. Garcia-Molina and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. in *Data Engineering, 2002. Proceedings. 18th International Conference on*. 2002: IEEE.
 - [52] Foggia, P., G. Percannella and M. Vento, Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 2014. 28(01).
 - [53] Chan, P.P.F. and C. Collberg. A Method to Evaluate CFG Comparison Algorithms. in *Quality Software (QSIC), 2014 14th International Conference on*. 2014.
 - [54] Almer, O., et al. An end-to-end design flow for automated instruction set extension and complex instruction selection based on GCC. in *Proceedings of the First International Workshop on GCC Research Opportunities (GROW'09)*. 2009.
 - [55] Bruschi, D., L. Martignoni and M. Monga, Detecting self-mutating malware using control-flow graph matching, in *Detection of Intrusions and Malware & Vulnerability Assessment*. 2006, Springer. p. 129-143.
 - [56] 吴健, 阮园与王少培, CoSyC 语言编译器安全性研究. *计算机工程*, 2012. 38(6): 第 43-46 页.
 - [57] Ullmann, J.R., An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 1976. 23(1): p. 31-42.
 - [58] Cordella, L.P., et al. Fast graph matching for detecting CAD image components. in *Pattern Recognition, 2000. Proceedings. 15th International Conference on*. 2000: IEEE.
 - [59] Cordella, L.P., et al., A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2004. 26(10): p. 1367-1372.
 - [60] McKay, B.D., Practical graph isomorphism. 1981: Department of Computer Science, Vanderbilt University.
 - [61] Fu, J.J., Directed graph pattern matching and topological embedding. *Journal of Algorithms*, 1997. 22(2): p. 372-391.
 - [62] Kawahito, M., et al., Idiom recognition framework using topological embedding. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013. 10(3): p. 13.
 - [63] Kawahito, M., Method, computer program and computer system for assisting in analyzing program. 2014, Google Patents.
 - [64] Conte, D., et al., A comparison of explicit and implicit graph embedding methods for pattern recognition, in *Graph-Based Representations in Pattern Recognition*. 2013, Springer. p. 81-90.
 - [65] Cooper, K.D., T.J. Harvey and T. Waterman, Building a control-flow graph from scheduled assembly code. Dept. of Computer Science, Rice University, 2002.
 - [66] Dullien, T. and S. Porst, REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.
 - [67] Aho, A.V., R. Sethi and J.D. Ullman, Compilers: principles, techniques, and tools. 1986: Addison-Wesley Longman Publishing Co., Inc. 796.
 - [68] Jensen, S., A. Møller and P. Thiemann, Type Analysis for JavaScript. 2009. 5673: p. 238-255.
 - [69] Davidson, J.W. and D.B. Whalley, Quick compilers using peephole optimization. *Softw. Pract. Exper.*, 1989. 19(1): p. 79-97.

-
- [70] Schkufza, E., R. Sharma and A. Aiken. Stochastic superoptimization. in ACM SIGARCH Computer Architecture News. 2013: ACM.
 - [71] Gibbons, P.B. and S.S. Muchnick, Efficient instruction scheduling for a pipelined architecture. SIGPLAN Not., 1986. 21(7): p. 11-16.
 - [72] Tip, F., A survey of program slicing techniques. Journal of programming languages, 1995. 3(3): p. 121-189.
 - [73] Mooney, C.H. and J.F. Roddick, Sequential pattern mining -- approaches and algorithms. ACM Comput. Surv., 2013. 45(2): p. 1-39.
 - [74] Ayres, J., et al. Sequential PAttern mining using a bitmap representation. in Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. 2002. Edmonton, Alberta, Canada: ACM.
 - [75] Zou, J., et al. Frequent instruction sequential pattern mining in hardware sample data. in Data Mining (ICDM), 2010 IEEE 10th International Conference on. 2010: IEEE.
 - [76] Cifuentes, C., et al., The university of queensland binary translator (uqbt) framework. The University of Queensland, Sun Microsystems, Inc, 2001.
 - [77] Terei, D.A. and M.M.T. Chakravarty, An LLVM Backend for GHC. Acm Sigplan Notices, 2010. 45(11): p. 109-120.
 - [78] Zhao, J., et al., Formalizing the LLVM intermediate representation for verified program transformations. In 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL, 2012. 47(1): p. 427-439.
 - [79] Chipounov, V. and G. Candea, Dynamically Translating x86 to LLVM using QEMU. 2010.
 - [80] Zynamics. BinNavi Product Homepage. <http://www.zynamics.com/binnavi.html> . 2015 March.
 - [81] Cytron, R., et al. An efficient method of computing static single assignment form. in Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1989: ACM.
 - [82] Briggs, P., et al., Practical improvements to the construction and destruction of static single assignment form. Software-Practice and experience, 1998. 28(8): p. 859-882.
 - [83] Appel, A.W., SSA is functional programming. SIGPLAN notices, 1998. 33(4): p. 17-20.
 - [84] Grover, V., A. Kerr and S. Lee. PLANG: PTX Frontend for LLVM. in LLVM Developers' Meeting. 2009.
 - [85] p2p Zeus. [http://en.wikipedia.org/wiki/Zeus_\(malware\)](http://en.wikipedia.org/wiki/Zeus_(malware)) . 2015 March.
 - [86] Proebsting, T.A. and S.A. Watterson, Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?)., in COOTS. 1997. p. 185-198.
 - [87] F.L.I.R.T In-depth. https://www.hex-rays.com/products/ida /tech/flirt/in_depth.shtml. 2015 March.
 - [88] Cong, J., H. Huang and W. Jiang. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010. 2010.
 - [89] Luo, L., et al. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014. Hong Kong, China: ACM.
 - [90] Kruegel, C., et al., Polymorphic Worm Detection Using Structural Information of Executables. 2006. 3858: p. 207-226.
 - [91] Warren, H.S., Hacker's delight. 2012: Pearson Education.
 - [92] Authors, U. LAPIN-SPAM: An Improved Algorithm for Mining Sequential Pattern. in 2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW). 2005.

- [93] Liu Zhangpei. xml test suite. <https://github.com/livenowhy/xml>. 2015 March.
- [94] Aho, A.V., et al., The AWK programming language. 1988: Addison-Wesley New York.
- [95] Ted Nyman. awk test suite. <https://github.com/tnm/awk>. 2015 March.
- [96] Coapp-packages. grep test suite. <https://github.com/coapp-packages/grep> . 2015 March.
- [97] sqlite test suite. <http://www.sqlite.org/download.html>. 2015 March.
- [98] Chipounov, V. and G. Candea. Enabling sophisticated analyses of $\times 86$ binaries with RevGen. in Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on. 2011: IEEE.
- [99] REC Decompiler. <http://www.backerstreet.com/rec/rec.htm>. 2015 March.
- [100] Luk, C., et al. Pin: building customized program analysis tools with dynamic instrumentation. in Acm Sigplan Notices. 2005: ACM.
- [101] Nethercote, N. and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. in ACM Sigplan notices. 2007: ACM.
- [102] Cavazos, J. and M.F. O'Boyle. Automatic tuning of inlining heuristics. in Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference. 2005: IEEE.
- [103] Lokuciejewski, P., et al. Automatic WCET reduction by machine learning based heuristics for function inlining. in 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART). 2009.

攻读学位期间取得的研究成果

- [1] Liu Bin, Zhao Yinliang, Li Meirong, Liu Yanzhao and Feng Boqin. A Virtual Sample Generation Approach for Speculative Multithreading Using Feature Sets and Abstract Syntax Trees. in Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on. 2012.
- [2] 刘延昭, 赵银亮, 武万杰. PowerPC 汇编程序反编译研究[J]. 计算机技术与发展.

学位论文独创性声明（1）

本人声明：所呈交的学位论文系在导师指导下本人独立完成的研究成果。文中依法引用他人的成果，均已做出明确标注或得到许可。论文内容未包含法律意义上已属于他人的任何形式的研究成果，也不包含本人已用于其他学位申请的论文或成果。

本人如违反上述声明，愿意承担以下责任和后果：

1. 交回学校授予的学位证书；
2. 学校可在相关媒体上对作者本人的行为进行通报；
3. 本人按照学校规定的方式，对因不当取得学位给学校造成的名誉损害，进行公开道歉。
4. 本人负责因论文成果不实产生的法律纠纷。

论文作者（签名）： 日期： 年 月 日

学位论文独创性声明（2）

本人声明：研究生 所提交的本篇学位论文已经本人审阅，确系在本人指导下由该生独立完成的研究成果。

本人如违反上述声明，愿意承担以下责任和后果：

1. 学校可在相关媒体上对本人的失察行为进行通报；
2. 本人按照学校规定的方式，对因失察给学校造成的名誉损害，进行公开道歉。
3. 本人接受学校按照有关规定做出的任何处理。

指导教师（签名）： 日期： 年 月 日

学位论文知识产权权属声明

我们声明，我们提交的学位论文及相关的职务作品，知识产权归属学校。学校享有以任何方式发表、复制、公开阅览、借阅以及申请专利等权利。学位论文作者离校后，或学位论文导师因故离校后，发表或使用学位论文或与该论文直接相关的学术论文或成果时，署名单位仍然为西安交通大学。

论文作者（签名）： 日期： 年 月 日

指导教师（签名）： 日期： 年 月 日

(本声明的版权归西安交通大学所有，未经许可，任何单位及任何个人不得擅自使用)