面向 iOS 移动应用的切片工具设计与实现

- 技术报告 -

Janus 研究中心
上海F众信息技术有限公司

朱冠杰
<anonymous@pwnzen.com>

谢斯坷夫
<anonymous@pwnzen.com>

唐祝寿
<anonymous@pwnzen.com>

本项目受国家互联网应急中心（CERT）资助。

目录

序言

iPhone 做为 *iOS* 应用的载体占据移动设备市场很大的份额。从 *2008* 年 *iPhone* 应用发布平台 *App Store* 正式上线至今，应用数量已经由最初的 *500* 款增长至目前的逾 *200* 万款。然而，针对 *iOS* 应用系统化的安全评估工作始终没有展开。*iOS* 原生系统的安全性、应用生态的封闭性、应用上架审核机制等，都从一定程度上缓解了 *iOS* 应用的安全威胁，但这并不能说明 *iOS* 应用不存在安全问题。目前针对 *iOS* 应用无现成的自动化检测工具，而逐渐增加的应用规模和复杂度阻碍了大规模的人工审计。安全研究社区目前亟需一款针对 *iOS* 应用安全的自动化检测工具。针对该问题，本工作对现有分析框架对进行了改进，通过将 *iOS* 应用机器码（*ARMv8*）转化为 *LLVM IR*，并在 *LLVM IR* 的基础上实施程序切片。在该工具基础上，结合其他分析技术，对 *iOS* 应用进行了安全研究。

本报告是 *Janus* 研究中心在 *iOS* 应用安全检测工作的一部分内容，报告标题、内容都将随着研究的不断深入，内容的不断扩展而变化。

Chapter 1

简介

iOS 应用切片工具分为两大部分，分别为：翻译模块和程序切片模块。翻译模块对 iOS 应用的二进制代码进行递归遍历来解码，并基于解码结果生成语意等价的 LLVM IR。在 LLVM IR 的基础上，进行指针分析，并开展切片工作。工具的具体工作流程如图1.1表示。

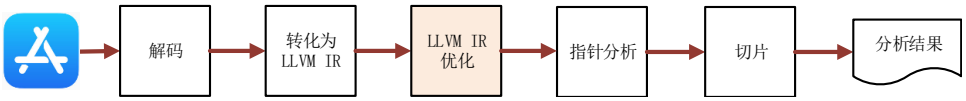


Figure 1.1: 静态切片工具流程图

Figure 1.1: Workflow of the static slicer

本章分节介绍该工具的各个模块，在介绍相关程序分析技术后，对比分析各种技术的开销，基于分析结果，提出优化措施和业内最佳实践。开销分析是基于 6 个随机挑选的中型 iOS 应用进行的，这 6 个应用程序分别为：NOW、京东读书、QQ 浏览器、Skype、NetFlix、Dancing Line。

Chapter 2

背景

2.1 Objective-C/SWIFT 简介

iOS 应用官方开发语言有两种：Objective-C 和 Swift。Objective-C（OC）由 Smalltalk 语言演化而来，是一种消息传递型（Message Dispatch）语言。在 OC 语言模型中，OC 类又称为对象，如代码2.1中的 GCDWebServer 类对象。为了实现方法调用，开发者需要指定类对象和方法，又叫做接收者对象（receiver）和方法选择器（selector），对应代码2.1第3行中的 GCDWebServer 和 alloc 等。

Listing 2.1: Objective-C 代码

```
1  - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
2      // Override point for customization after application launch.
3      GCDWebServer* webServer = [[GCDWebServer alloc] init];
4      [webServer addGETHandlerForBasePath:@"/" directoryPath:NSHomeDirectory() indexFilename:nil cacheAge:3600
           allowRangeRequests:YES];
5      [webServer start];
6
7      return YES;
8  }
```

为通过消息传递的方式调用目标函数，编译器将 OC 代码编译成针对运行期库的调用，如代码 2.2中第11行所示的 _objc_msgSend，然后通过运行期库将函数调用分发到真实的目标函数中。该语言的动态绑定（dynamic binding）特性决定了编译器并不关心接收消息的对象是什么类型，在运行时才会去查找真正所要调用的方法，接收消息的对象也是在运行时才工作。

Listing 2.2: OC 代码2.1的部分反汇编结果

```
...                                                                                                                    1
__text:0000000100006794          ADRP          X8, #selRef_alloc@PAGE                                                  2
__text:0000000100006798          ADD           X8, X8, #selRef_alloc@PAGEOFF                                           3
__text:000000010000679C          ADRP          X9, #classRef_GCDWebServer@PAGE                                         4
__text:00000001000067A0          ADD           X9, X9, #classRef_GCDWebServer@PAGEOFF                                  5
__text:00000001000067A4          MOV           X0, #0                                                                  6
__text:00000001000067A8          STR           X0, [SP,#0x50+var_28]                                                   7
__text:00000001000067AC          LDR           X9, [X9] ; _OBJC_CLASS_$_GCDWebServer                                   8
__text:00000001000067B0          LDR           X1, [X8] ; "alloc"                                                      9
__text:00000001000067B4          MOV           X0, X9 ; void *                                                         10
__text:00000001000067B8          BL            _objc_msgSend                                                           11
__text:00000001000067BC          ADRP          X8, #selRef_init@PAGE                                                   12
__text:00000001000067C0          ADD           X8, X8, #selRef_init@PAGEOFF                                            13
__text:00000001000067C4          LDR           X1, [X8] ; "init"                                                       14
__text:00000001000067C8          BL            _objc_msgSend                                                           15
__text:00000001000067CC          LDR           X8, [SP,#0x50+var_28]                                                   16
__text:00000001000067D0          STR           X0, [SP,#0x50+var_28]                                                   17
__text:00000001000067D4          MOV           X0, X8                                                                  18
__text:00000001000067D8          BL            _objc_release                                                           19
__text:00000001000067DC          LDR           X8, [SP,#0x50+var_28]                                                   20
__text:00000001000067E0          STR           X8, [SP,#0x50+var_38]                                                   21
__text:00000001000067E4          BL            _NSHomeDirectory                                                        22
__text:00000001000067E8          MOV           X29, X29                                                                23
__text:00000001000067EC          BL            _objc_retainAutoreleasedReturnValue                                     24
__text:00000001000067F0          ADRP          X8, #stru_100008060@PAGE ; "/"                                          25
__text:00000001000067F4          ADD           X8, X8, #stru_100008060@PAGEOFF ; "/"                                   26
__text:00000001000067F8          MOV           X9, #0                                                                  27
__text:00000001000067FC          MOV           X5, #0xE10                                                              28
__text:0000000100006800          MOV           W10, #1                                                                 29
__text:0000000100006804          ADRP          X1,                                                                     30
      #selRef_addGETHandlerForBasePath_directoryPath_indexFilename_cacheAge_allowRangeRequests_@PAGE
__text:0000000100006808          ADD           X1, X1,                                                                 31
      #selRef_addGETHandlerForBasePath_directoryPath_indexFilename_cacheAge_allowRangeRequests_@PAGEOFF
__text:000000010000680C          LDR           X1, [X1] ; "addGETHandlerForBasePath:directoryPath:"...                 32
__text:0000000100006810          LDR           X2, [SP,#0x50+var_38]                                                   33
__text:0000000100006814          STR           X0, [SP,#0x50+var_40]                                                   34
__text:0000000100006818          MOV           X0, X2 ; void *                                                         35
__text:000000010000681C          MOV           X2, X8                                                                  36
__text:0000000100006820          LDR           X3, [SP,#0x50+var_40]                                                   37
__text:0000000100006824          MOV           X4, X9                                                                  38
__text:0000000100006828          AND           W6, W10, #1                                                             39
__text:000000010000682C          BL            _objc_msgSend                                                           40
__text:0000000100006830          LDR           X0, [SP,#0x50+var_40]                                                   41
__text:0000000100006834          BL            _objc_release                                                           42
__text:0000000100006838          ADRP          X8, #selRef_start@PAGE                                                  43
__text:000000010000683C          ADD           X8, X8, #selRef_start@PAGEOFF                                           44
__text:0000000100006840          LDR           X9, [SP,#0x50+var_28]                                                   45
__text:0000000100006844          LDR           X1, [X8] ; "start"                                                      46
__text:0000000100006848          MOV           X0, X9 ; void *                                                         47
__text:000000010000684C          BL            _objc_msgSend                                                           48
...                                                                                                                    49
__text:00000001000068A0          RET                                                                                   50
__text:00000001000068A0 ; End of function -[AppDelegate application:didFinishLaunchingWithOptions:]                   51
```

OC 的这种函数调用方式导致了分析上的困难，如在代码 2.2中，为了确定第11行 _objc_msgSend 的调用实际被分发到针对 GCDWebServer 类对象中的 alloc 方法的调用，需要识别 X0 号寄存器和 X1 号寄存器所指向的内容。通过程序分析，恢复函数调用的语意信息，即本文切片方面的主要工作。

2.2 iOS 应用文件简介

iOS 应用是一个以.ipa 结尾的压缩文件，文件中包含 Info.plist 文件、可执行代码文件、资源文件和其他支撑文件。Info.plist 文件以“键-值”对的形式对信息进行存储，如其中的 CFBundleIdentifier 字段指定 Bundle ID，CFBundleExecutable 字段指定可执行代码文件。

iOS 应用可执行代码以 Mach-O 文件格式进行组织 [26]，一个 Mach-O 文件由三部分组成，分别为：

- Header：包含 Mach-O 文件的标识符和运行目标平台等信息；
- Load Commands：指定文件布局和内存中的段地址；
- Data：指定可以加载到内存的节和区域。

Mach-O 文件的简单结构如图 2.1所示。其中，Data 域中的节包含了应用执行所需要的全部信息，这些节有：

- ___text：应用中的代码，用 ARM/THUMB 指令集编码；
- ___cstring：应用中的字符串；
- ___objc_classlist：指向类的指针列表，这些指针指向 ___objc_data 节；
- ___objc_data：包含了指向超类和指向位于 ___objc_const 节的类信息的指针；
- ___objc_const：包含每个类的细节，类中的方法、协议、实例变量和一些属性信息；
- Dynamic Loader Info：包含了文件中未出现的类，比如针对 Objective-C 运行期库的引用等信息。通过遍历这个结构，可以重建整个类层次结构。

2.3 LLVM 简介

LLVM [22, 42] 是由一系列编译器和工具链（反汇编、编译、调试器等）集合组成的框架，框架通过模块化设计做到可重用。LLVM 整体结构如图2.2所示。

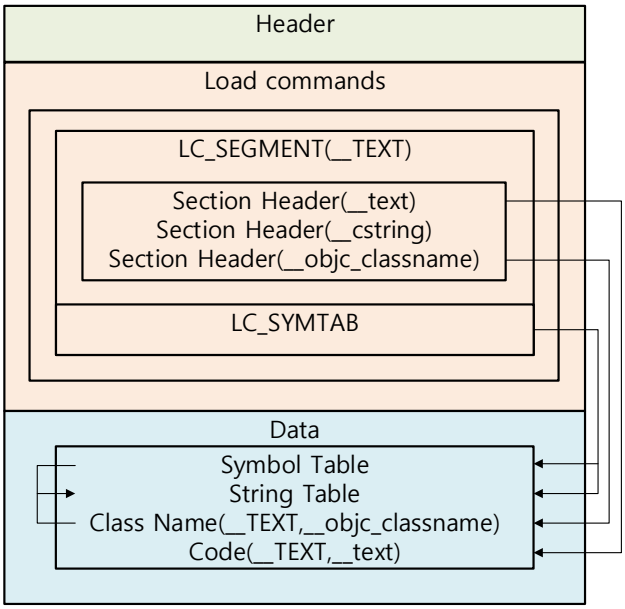


Figure 2.1: Mach-O 文件结构

Figure 2.1: The simplified inner structure of Mach-O file.

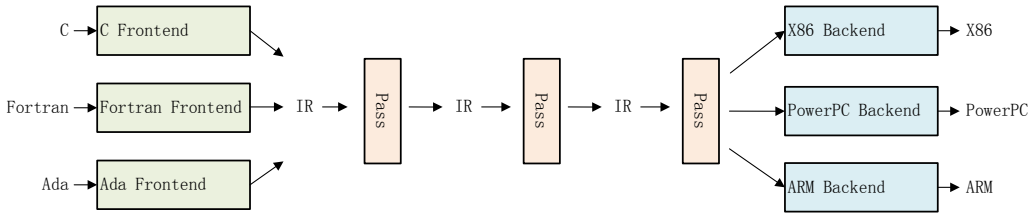


Figure 2.2: LLVM 架构

Figure 2.2: LLVM’s architecture

LLVM 分前端（front end）、优化器和后端（back end）三个部分，前端用来解析不同编程语言，优化器围绕中间表示（IR）做优化工作，后端针对特定的 CPU 生成指令。

LLVM 作为一个编译器，其前端接收源代码后，在分词（tokenized）、解析（parsed）和分析（analyzed）后，生成 LLVM IR。其后端负责机器指令翻译、链接目标程序等任务，在这个过程中，LLVM 后端针对特定 CPU 的寄存器、指令集输出代码[1]。

LLVM IR 通过 Module、Function、BasicBlock 和 Instruction 的方式组织。其中 Module 包含 Function，Function 包含 BasicBlock，BasicBlock 包含 Instruction。

[1] http://llvm.org/docs/CodeGenerator.html

其包含关系如图 2.3所示。

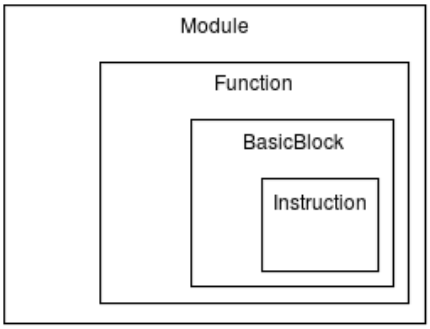


Figure 2.3: LLVM IR 结构

Figure 2.3: Layout of LLVM IR

LLVM Pass 框架是 LLVM 系统中的一个重要组成。Pass 负责 LLVM 系统中的转换和优化工作。开发者结合要分析的对象，可以继承不同的类（如 ModulePass、LoopPass 或 BasicBlockPass 等）来实现自己的 Pass。如开发者可以编写从 ModulePass 派生的 Pass 以分析整个程序，这个派生的 Pass 可以乱序对整个程序的函数进行引用，或添加和删除程序中的函数等。除了用户可以自行编写 Pass，LLVM 本身提供其他一些（110 个左右）实用的 Pass 用于分析、转化等工作[2]，本工作所用到的内置 Pass 包括：

- Post-Dominator Tree Construction（-postdomtree: ）后序支配树；
- Post-Dominance Frontier Construction（-postdomfrontier: ）后序支配边界；
- Natural Loop Information（-loops: ）识别自然循环；
- Dominator Tree Construction（-domtree:）：前序支配树；
- Scalar Replacement of Aggregates（-sroa: ）：结构和数组的标量替换；
- Combine redundant instructions（-instcombine: ）：冗余指令合并；
- Deduce function attributes（-functionattrs: ）：函数属性推导；
- Promote Memory to Register（-mem2reg:）：内存引用提升为寄存器引用；
- Simple mod/ref analysis for globals（-globalsmodref-aa: ）全局变量修改/引用分析；
- Simplify the CFG（-simplifycfg:）死代码删除和基本块合并；
- Simple constant propagation （-constprop: ）常量传播；

[2] https://llvm.org/docs/Passes.html

2.4 TableGen 简介

LLVM 后端需要使用目标架构详细的描述信息以生成目标代码。该描述的信息量很大，而且容易编写出错。为了减轻工作量，降低复杂度，LLVM 规定了目标架构信息的描述语言（TableGen 语言），并在工具链中提供工具（如 llvm-tblgen[3]）来解析这些语言（.td 文件）。

TableGen 语言包含定义、类、多类（multiclass）等几个关键部分[4]，这几个部分最终会被扩展成记录（records）。在词法上，TableGen 语言定义了完整的词法规范，包括标点符号、关键字、运算符等[5]。在语法上，TableGen 语言约定了基本的元语（类型、值、表达式等）、类的定义和文件包含等规范[6]。在语法风格上，TableGen 语言中类的定义与 C++ 语言的模版类类似，类的支持为通用信息描述提供了可能，在很大程度上降低了描述信息的复杂程度。TableGen 后端解析基于 TableGen 语言编写的描述文件，以代码的形式输出解析结果[7]，这些代码对目标架构进行了实际解析。TableGen 具体的工作流程如图 2.4所示。

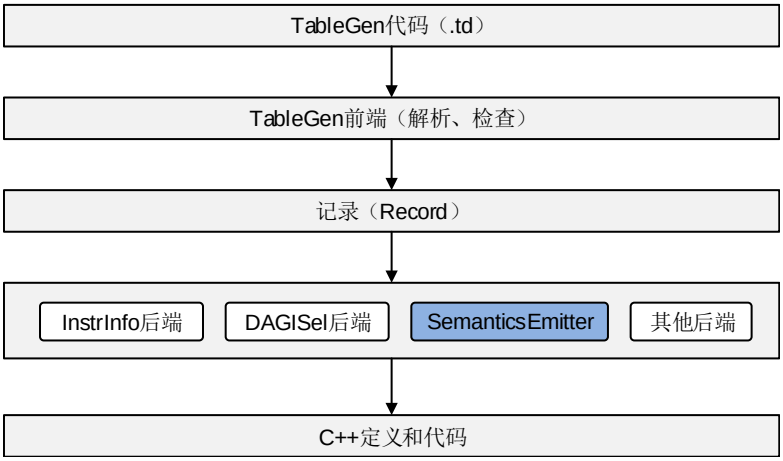


Figure 2.4: TableGen 工作流程

Figure 2.4: Workflow of tablegen

[3] https://llvm.org/docs/CommandGuide/tblgen.html

[4] https://llvm.org/docs/TableGen/index.html

[5] https://llvm.org/docs/TableGen/LangRef.html

[6] https://llvm.org/docs/TableGen/LangIntro.html

[7] https://llvm.org/docs/TableGen/BackEnds.html

2.5 Clang 简介

LLVM 是苹果公司主导的开源框架，在 LLVM 后端的基础上，苹果开发了前端框架 Clang，并在后端增加了开源框架以外的一些额外指令的支持。苹果提供的开发环境 XCode 集成了 Clang，因此，iOS 应用都是由 Clang 编译生成。

Clang 支持 C/C++/Objective-C/SWIFT 语言，除了常规的词法、语法等常规前端功能外，还提供自动引用计数功能（Automatic Reference Counting，ARC）[8]，通过该特性减轻开发者内存管理的负担。在编译 iOS 应用时，如果开发者开启 ARC 功能支持，相关 ARC 功能代码自动插入到目标文件中。

结合本文优化需求，将这些 ARC 代码分为 2 类，分别为语意无关的 ARC 代码和语意相关的 ARC 代码。如 ARC 运行期接口：

```
id objc_autoreleaseReturnValue(id value);
```

的调用总是返回 value，该调用只是向 ARC 运行期注册以进行内存的自动管理，删除该调用并不会影响程序的语意，因此将其称为语意无关的 ARC 代码。而下面接口：

```
void objc_copyWeak(id *dest, id *src);
```

除内存管理注册外，还包含对象拷贝的动作。因此我们将其定义为语意相关的 ARC 代码。

[8] https://clang.llvm.org/docs/AutomaticReferenceCounting.html

Chapter 3

iOS 应用二进制代码解码

为了能对 iOS 应用进行自动化分析，先对 iOS 应用二进制代码进行解码（反汇编），解码的对象为Mach-O文件格式中的ARMv8指令。解码的过程实际是 ARMv8 指令的翻译过程，该过程从二进制代码中恢复 ARMv8 指令层面的语意信息。在实际操作中，通过在LLVM 后端实现一个基于 LLVM MC 的递归解码模块来实现，与此同时，添加数据结构来表示指令层面的控制流图。

3.1 相关背景

3.1.1 ARMv8 架构

ARMv8 是 ARMv7 之后的一个重要架构更新。ARMv8 的一个主要变化是引入了 64 位的架构，即 AArch64。在 AArch64 状态下执行的代码只能使用 A64 指令集。为了向后兼容，ARMv8 保持与现有 32 位体系结构兼容的 AArch32，即 AArch64 之前的指令也得到支持。第一套 ARMv8 架构的处理器产品为 Cortex-A50 系列，包含 Cortex-A53、Cortex-A57 两个产品型号。Cortex-A50 系列均集成了 NEON SIMD 引擎、ARM CoreSight 多核心调试与追踪模块、128-bit AMBA ACE 一致性总线界面，还可选加密加速单元。在 LLVM 的目标架构描述文件中，对 ARMv8 架构有相关的描述（AArch64.td 等）。

iOS 可执行代码被编译成针对某种 CPU 结构的代码。从 2013 年开始，iOS 设备（iPhone 5s）使用了 64 比特的 ARMv8 CPU，目前的 iOS 应用代码都适配此结构。为了兼容，iOS 可执行代码（Universal Binaries）通常同时支持 AArch32 和 AArch64 架构指令集，相关代码分别由Mach-O文件的头部指定。LLVM 的工具链中提供 llvm-

lipo [1]工具对相关架构代码进行抽取，因此，分析框架只需提供针对 AArch64 架构指令集的分析即可。

3.1.2 现有的反汇编工具

目前能够对 iOS 应用二进制代码反汇编的工具包括：objdump [2]、IDA [3]、Hopper [4]、Ghidra [5] 等。另外，LLVM MC [6]、CapStone [7] 等提供相关的 API，可对指定的机器码进行反汇编。其中，objdump 等属于线性反汇编工具，这类工具无法识别代码节区中的数据信息，因此，如果代码节区中有数据信息时，这类工具无法正常反汇编，导致反汇编失败或在语义上产生错误。IDA、Hopper 等工具，使用的是迭代反汇编方法，能有效区分代码和数据，但这类软件属于独立商业软件，其他分析工具的介入难度较高，无法开展复杂分析。LLVM MC、CapStone 等属于基本的反汇编框架，可对单条指令进行反汇编，需要其他工具的指引才能进行反汇编。

以上工具都追求忠实的反汇编，其结果更适合人工审阅，而不能用于自动化分析。为了适应自动化分析的需求，目前有一些基于 LLVM 的框架将 Mach-O 格式可执行文件转化为适合程序分析的 IR，这些框架包括：dagger [8]和 retdec [9]等，而我们的工作即基于 dagger 进行的。

3.2 iOS 二进制代码解码实现

iOS 二进制代码解码模块在 LLVM 的后端实现。通常情况下，LLVM 前端负责生成目标无关的 IR，后端负责将 LLVM IR 转化为适配目标架构的指令/操作数。为了完成 iOS 二进制代码解码，本工作在后端实现了通常在前端完成的工作，主要的原因是在后端进行解码可以充分利用TableGen所带来的便利。

LLVM 框架的 MC 层提供机器码的翻译功能，且 LLVM 内置了针对 AArch64 的描述文件，如机器指令描述、寄存器描述和微架构的调度描述等。使用 llvm-tblgen

[1] llvm-lipohttps://llvm.org/docs/CommandGuide/llvm-lipo.html
[2] https://llvm.org/docs/CommandGuide/llvm-objdump.html
[3] https://www.hex-rays.com/
[4] https://www.hopperapp.com/
[5] https://ghidra-sre.org/
[6] https://llvm.org/docs/CodeGenerator.html#the-mc-layer
[7] http://www.capstone-engine.org/
[8] https://github.com/repzret/dagger
[9] https://github.com/avast/retdec

工具，可以将这些描述文件转化为代码框架，框架指定处理方法来翻译指令、操作数，这些方法最终用来生成反汇编代码。该过程体现在算法 1第 7行。

Mach-O 文件中已经存储了 Objective-C 函数的信息，包括函数名称、函数在二进制文件中的地址等。以这些函数为目标，进行二进制代码的解码。鉴于线性解码将导致代码中的数据（如位于代码节区中跳转表、填充数据等）都被解码，为生成语意一致的解码结果，解码模块以 Objective-C 函数的入口点为起始点，逐条对指令解码。解码过程中遇到分支指令 'B'，'BL' 等，生成基本块（BB），遇到函数调用指令时，生成 C/C++ 函数。完整的解码过程如算法 1所示。经过这样一个过程，建立了反汇编层面的函数、基本块、指令。

Algorithm 1 迭代解码

Input: 方法列表（*m_list*）

```
1: for m ∈ m_list do
2:     for i ∈ m.binary_list do
3:         if i = m.start_addr then
4:             genMCFunction(i)
5:             genMCBasicBlock(i)
6:         end if
7:         MCInst ← getInstruction(i)
8:         if MCInst ∈ B_INST then
9:             if MCInst ∉ C_INST then
10:                if getTarget(MCInst) then
11:                    genMCBasicBlock(MCInst)
12:                end if
13:                genMCBasicBlock(i.addr + instSize)
14:            end if
15:        end if
16:    end for
17: end for
```

3.3 iOS 二进制代码解码优化

如表 3.1第 2 列所示，一般应用往往包含百万级的指令，这为后续向 LLVM IR 转化带来时间和空间上的压力。采用迭代解码的方法在确保语意正确的前提下，在一定程度上也降低了不必要的计算和存储开销，如表 3.1第 2、3 列所示。在此基础上，

Table 3.1: 指令优化结果

Table 3.1: Instruction optimize result of the disassembler

应用名称	线性指令[a]	迭代指令[b]	移除指令[c]	ARC 调用[d]	压缩比[e]	时间[f] (s)	内存[f] (M)
NOW	9,623,039	8,826,556	305,176	424,740	8.27%	75.8683	3883.11
京东读书	3,340,350	3,217,610	101,513	229,905	10.30%	15.3931	1396.44
QQ 浏览器	9,197,941	8,793,586	379,734	576,491	10.87%	81.3502	3843.69
Skype	5,175,657	4,872,568	142,372	59,546	4.14%	30.6160	2171.48
NetFlix	3,361,059	3,120,760	53,275	133,551	5.99%	14.0862	1348.48
DancingLine	8,172,013	7,815,551	514,497	228,882	9.51%	66.6102	4965.18
均值	6,478,343	6,107,772	249,428	275,519	8.59%	47.3207	2934.73

[a] 代码节区中线性指令的数量

[b] 实际迭代反汇编识别的指令数量（IC）

[c] 实际迭代反汇编识别的非通用指令的数量（NONE-GENERAL）

[d] 实际迭代反汇编识别的 ARC 方法调用的数量（ARC）

[e] (NONE-GENERAL + ARC) / IC

[f] 测试机：MacBook Pro (Retina, 15-inch, Mid 2015)；CPU：2.5 GHz Intel Core i7；内存：16 GB 1600 MHz DDR3

我们做了进一步的优化工作，包括语意无关的优化和语意相关的优化。经过这些优化，又去除了 8.59% 的指令。

3.3.1 指令删除优化

语意无关的优化：语意无关的优化即删除该指令并不影响原程序的语意。这部分的工作主要针对编译器 Clang 插入的引用计数器（ARC）进行的。通过识别语意无关的 ARC 调用（如 *__objc_release*），来减少指令的数量。优化结果如表 3.1的第 5 列所示。

语意相关的优化：语意相关的优化是指删除该指令会影响原程序的语意。但后续的程序分析对该语意不敏感，因此可以在反汇编过程中删除。如结合待分析的问题，我们发现浮点数、向量运算等不会对分析结果产生影响，因此我们将浮点数、向量运算等指令删除，优化结果如表 3.1的第 4 列所示。

Table 3.2: 操作数长度统计

Table 3.2: Number of operands

操作数长度（#）	1	2	3	4	5	6	7	8
NOW	1,595,283	1,275,395	2,330,031	3,556,600	314,533	900	0	0
京东读书	656,429	484,491	776,473	1,319,740	108,109	11	0	0
QQ 浏览器	1,747,500	1,344,913	2,141,654	3,537,752	318,407	326	0	0
Skype	688,876	631,207	1,536,875	1,969,861	146,267	412	0	0
NetFlix	538,502	443,643	859,176	1,292,504	71,779	0	0	0
DancingLine	1,438,337	1,048,581	2,539,612	2,854,412	258,372	80	0	0

Table 3.3: Skype 内存开销优化结果

Table 3.3: Overhead of optimize result for Skype

操作数数量（#）	1	2	3	4	5	6	7	8
内存开销（M）	1629.45	1733.46	1824.56	1709.12	1902.44	1975.68	2096.70	2171.48
时间开销（s）	404.3	364.1	353.0	366.7	414.1	410.8	391.4	370.9

3.3.2 内部存储优化

由表 3.1第 7、8 列可见，即使在指令上进行了优化，反汇编在时间/空间上依然有很高的开销。原始 dagger 在内部存储上并没有过多考虑开销的问题，如 dagger 为指令的操作数设定了长度为 8 的固定存储以加快存取速度。为了降低解码在空间上的开销，我们针对这些数据结构进行了优化。以操作数存储优化为例，实际操作数长度分布如表 3.2所示。由表 3.2可见，大部分指令长度小于 6。进一步的测试表明，根据实际需要分配操作数几乎不影响程序的执行时间，测试结果见表 3.3。由表 3.3所示，在不影响程序执行效率的前提下，仅这一项优化即可将 Skype 的内存开销降至原始的 75.04%，DancingLine 的内存开销降至原始的 64.36%。

3.4 本章小结

iOS 二进制代码翻译工作是一项非常复杂的工作，除了需要熟悉目标架构、指令，还要使用 TableGen 语言准确描述目标架构。而 TableGen 语言又是一种规则描述语言，各种调试支持等都相对缺乏，这在一定程度上也增加了语言使用上的复杂性。iOS 二进制代码翻译为 IR 转化提供了基础支撑，考虑到 IR 优化的复杂性，很多针对后续的优化工作都前移到这一步完成，比如为函数建立摘要等。

Chapter 4

转化为 LLVM IR

该过程将 iOS 应用解码结果转化为 LLVM IR，转化过程需要确保指令、控制流级别的语意一致，转化结果可以借助 LLVM 工具链中的一些分析工具来进行优化，后续的指针分析、切片分析也可以通过标准接口[1]来开展工作。

4.1 相关背景

4.1.1 LLVM IR

LLVM IR 是一种三地址码结构的语言，支持简单的加、减、比较等操作，同时又支持标签等语法，整体上与汇编语言类似，通过调用相关接口即可实现到 LLVM IR 的转化。与汇编不同的是，LLVM IR 是一种强类型语言，另外 LLVM IR 以静态单赋值（Static Single Assignment, SSA） [5] 形式表示，也就是说在 LLVM IR 中，有足够的寄存器可用[2]。除了语法上的定义，LLVM IR 提供内部结构以支持优化，提供 bitcode（.bc）方式以支持压缩存储[3]。生成 LLVM IR 后，通过 llvm-dis[4]工具可以观察其内部结构。如代码 4.2第1行表示将两个 64bit（类型）的值相加，其中一个值存放在*%SP_6* 寄存器（足够多的寄存器）中，另外一个值为立即数 8，相加的结果存贮在变量*%15* 中。

[1] https://llvm.org/docs/LangRef.html

[2] https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html

[3] https://blog.csdn.net/tristan_tian/article/details/81629849

[4] https://llvm.org/docs/CommandGuide/llvm-dis.html

4.2 LLVM IR 转化的实现

与 LLVM MC 类似，通过实现一个向 LLVM IR 转化的层来实现解码结果到 LLVM IR 的转化。在具体实现上，通过一个描述 AArch64 架构（包括指令、寄存器）语意信息的文件（.td）生成一个后端（SemanticsEmitter，见2.4），配合驱动代码和转化代码，完成翻译结果到 LLVM IR 的转化，生成符合规定格式的 LLVM IR，最终生成 LLVM IR 中包含函数、控制流、基本块等信息。

4.2.1 指令等价语意的转化

LLVM IR 一种目标无关的三地址码形式的中间表示，因此在转化时，需要将 AArch64 指令转化为语意等价的 LLVM IR。如代码 ARM 指令 4.1不是三地址码表示，该指令将被翻译成如代码 4.2所示 LLVM IR，其 SSA 风格的代码由 LLVM 自动生成。

Listing 4.1: 反汇编代码

```
1000060E4          LDR          X1, [SP,#8]
```

Listing 4.2: 反汇编代码 4.1的中间表示

```
[0x1000060E4] %15 = add i64 %SP_6, 8
[0x1000060E4] %16 = inttoptr i64 %15 to i64
[0x1000060E4] %X1_5 = load i64, i64* %16
```

4.2.2 寄存器的翻译

相比源代码，二进制层面的函数是没有参数信息的，参数往往是通过寄存器或者内存（最终还是通过寄存器来指向）来传递的。在向 LLVM IR 转化的过程中，所有函数都被翻译成只包含一个参数的函数，这个参数为代表上下文的全部寄存器。同时，因为返回值也包含在该上下文中，因此所有函数都被翻译成没有返回的函数，如下所示，其中*%regset* 的定义为代表全部寄存器的上下文。

define void @fn_100006290(%regset* noalias nocapture)

将函数转为 IR 时，为每个被使用的寄存器分配一个本地变量。在每个基本块内部，只能访问本地变量而非寄存器。一个基本块的开始部分从本地变量加载寄存

器值，结尾部分寄存器的值被保存到本地变量，基本块的中间部分采用的是 SSA 形式的表示。通过这种方式，可以使用Promote Memory to Register这个 Pass 来简化 SSA 中 ϕ 的表示[5]。

4.2.3 控制流的翻译

向 LLVM IR 翻译的过程中涉及跳转和调用相关的控制流。对于直接控制转化的翻译，通过解码 ARM 指令，在解码的过程中就能构建控制流。对于间接跳转、调用（*BR*、*BLR* 等通过寄存器传递目标地址），目标函数不确定，需要在后续分析模块中填充。

函数内部的基本块代表一种控制流程。跳转指令作为基本块 BB_1 结束的同时，也指向后续的基本块 BB_2。使用跳转指令的操作数可以精确确定后续基本块 BB_2 的入口点。基本块 BB_2 入口点的直接前驱也被设置成基本块 BB_1 的结束。从基本块的信息中，可以获取准确的函数内控制流图。

为了保存方法调用前的上下文信息，在函数调用前保存全部寄存器至上下文寄存器中，并且在调用后对其进行恢复，如代码 4.3所示。ARMv8 指令集中的函数调用（*BL* 和 *BLR*）被翻译成 LLVM IR 中的 *call* 指令（代码 4.3第14行），传递的实参被保存在寄存器与/或栈中，被调用函数通过寄存器或者间接使用寄存器来访问栈以获取参数[6]，因此这种表示方法在语意上是正确的。但使用这种方法，无法获取参数和返回值的信息，而这些信息对于重建过程间的数据流又是必须的，因此后续会对这些信息进行特殊处理。

Listing 4.3: 函数调用的中间表示

```
bb_1000060AC_call:                        ; preds = %bb_1000060AC
[0xFFFFFFFFF] store i64 %FP_5, i64* %FP_ptr
[0xFFFFFFFFF] store i64 %LR_init, i64* %LR_ptr
[0xFFFFFFFFF] store i64 %SP_6, i64* %SP_ptr
[0xFFFFFFFFF] store i64 %X0_6, i64* %X0_ptr
[0xFFFFFFFFF] store i64 %X2_init, i64* %X1_ptr
[0xFFFFFFFFF] store i64 %X2_init, i64* %X2_ptr
[0xFFFFFFFFF] store i64 %X3_init, i64* %X3_ptr
[0xFFFFFFFFF] store i64 %X4_init, i64* %X4_ptr
[0xFFFFFFFFF] store i64 %X5_init, i64* %X5_ptr
[0xFFFFFFFFF] store i64 %X6_init, i64* %X6_ptr
[0xFFFFFFFFF] store i64 %X7_init, i64* %X7_ptr
[0xFFFFFFFFF] store i64 %X8_init, i64* %X8_ptr
[0xFFFFFFFFF] call void objc_storeStrong(%regset* %0)
[0x100006114] %LR_5 = load i64, i64* %LR_ptr
[0x100006114] %X0_13 = load i64, i64* %X0_ptr
```

[5] https://www.cis.upenn.edu/~cis341/current/lectures/lec24.pdf

[6] http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf

```
[0x100006114] %X1_8 = load i64, i64* %X1_ptr
[0x100006114] %X2_2 = load i64, i64* %X2_ptr
[0x100006114] %X3_2 = load i64, i64* %X3_ptr
[0x100006114] %X4_1 = load i64, i64* %X4_ptr
[0x100006114] %X5_1 = load i64, i64* %X5_ptr
[0x100006114] %X6_1 = load i64, i64* %X6_ptr
[0x100006114] %X7_1 = load i64, i64* %X7_ptr
[0x100006114] %X8_5 = load i64, i64* %X8_ptr
[0xFFFFFFFFF] br label %bb_c1000060D4
```

4.2.4 外部符号重建

iOS 应用中为引用外部库的符号提供一些占位符。在运行期，这些占位符被外部库的信息填充。为能在静态分析阶段也能获取这些信息，通过模拟这个填充过程将符号名字和对应的库的信息进行恢复，如：

Change fn_10000632C to -[SceneDelegate scene:willConnectToSession:options:]

4.3 LLVM IR 转换过程优化

4.3.1 LLVM IR 翻译过程优化

语意相关的优化：由代码4.3可见，为了产生通用的 IR，每次函数调用都降创建大量的 LLVM IR 代码，虽然在后续的优化过程中，这些代码中的数据由于无依赖关系而会被优化，但是这些 IR 对转化过程产生了大量的影响。因此，在 LLVM IR 生成阶段，针对方法调用开展了相关的优化工作。对已知语意的方法，使用等价替换的方式来简化其 LLVM IR 表示。如针对ARC 相关调用，我们在解码阶段优化的基础上，继续对其进行优化以降低内存开销。该过程的优化原则为在不改变程序语意的情况下，用等价代码替换 ARC 相关调用，通过这种手段来降低 LLVM IR 在空间上的开销。如代码4.3将被替换为等价的 LLVM IR 语意的代码4.4，如4.4所示，替换后的代码从原始的 25 条语句减少为 2 条。

Listing 4.4: ARC 函数调用语意等价替换

```
[0x100006334] %X0_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 5
[0x100006334] %X0_init = load i64, i64* %X0_ptr
```

考虑到直接操作 LLVM IR 比较复杂（此时定值-使用链已经建立，SSA 形式代码已经生成），因此在实际操作中，我们将此阶段的优化工作前移，即在解码阶段，对这些指令进行等价替换。如代码4.3在解码阶段将被替换为等价的指令：

Table 4.1: IR 生成优化

Table 4.1: Optimization for LLVM IR generation

应用名称	内存开销（G）[a]	优化后内存（G）	时间开销（s）[ab]	优化后时间（s）[b]
NOW	12.00	9.55	310.0740	248.6394
京东读书	4.83	3.25	127.7634	84.3077
QQ 浏览器	12.59	9.27	373.8233	230.1499
Skype	5.92	5.58	153.1845	142.3253
NetFlix	4.05	3.28	109.2647	86.4530
DancingLine	10.87	9.58	274.7107	238.8626
均值	8.38	7.31	224.8034	171.7897

[a] 在解码优化的基础上的测试结果。

[b] 测试机：MacBook Pro (Retina, 15-inch, Mid 2015)；CPU：2.5 GHz Intel Core i7；内存：16 GB 1600 MHz DDR3。

100006334 LDR X0, [X0]

内部存储优化：与解码阶段的内部存储优化类似，对于 LLVM IR 代码的内部表示使用按需分配的原则，以降低其内存开销。

4.3.2 LLVM IR 外部优化

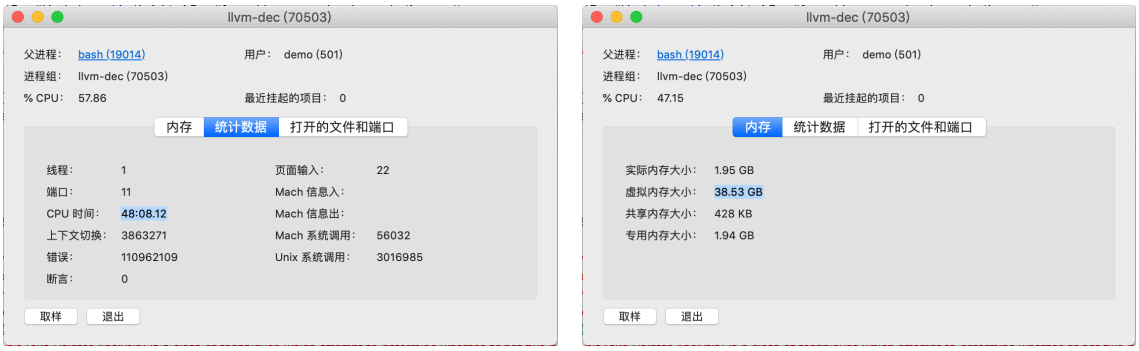
转化为 LLVM IR 的优势是可以使用 LLVM 工具链中的优化模块对 IR 进行优化，在 LLVM IR 转化结束后，使用 Combine redundant instructions、Scalar Replacement of Aggregates、Simplify the CFG、Simple constant propagation 几个模块对 IR 进行了优化。

4.4 LLVM IR 转化模块开销

LLVM IR 转化模块的开销如表 4.1所示，整体上内存降低至原始的 87.23%，时间上降低为原始的 76.42%。相比解码阶段的优化，LLVM IR 转化过程的优化效果不明显。

至此，将二进制代码转化为 LLVM IR。总体优化结果如图4.2和4.3所示。分析表明，针对ARC 的优化效果并不明显，优化主要通过解码阶段的指令删除优化和内部

存储优化来实现。从转化过程看，转化的开销与指令数量呈线性关系。但实际观测到，由于未优化，内存增长至系统极限后，指令转化速度开始下降，因为系统需要大量的时间来进行内存的压缩、置换，导致实际运行时间远远超过所需时间。因此，优化对系统性能的提升并非线性，如图4.1所示，优化前的 dagger 基本无法进行向 LLVM IR 转化的工作。



(a) 时间开销 **(b)** 内存开销

(a) Time overhead **(b)** Memory overhead

Figure 4.1: 原始 dagger 在分析 NOW 时的开销

Figure 4.1: Overhead of dagger when analyzing NOW app

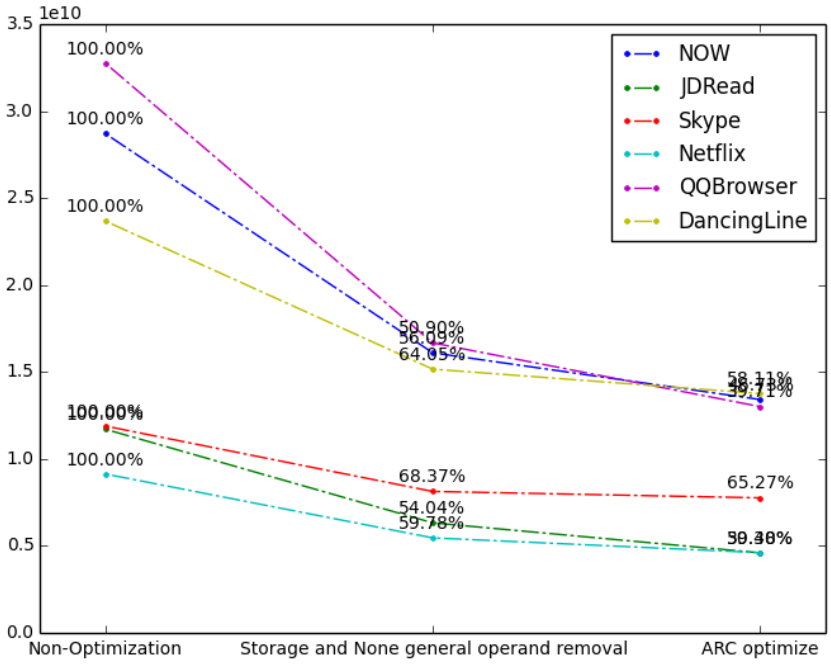


Figure 4.2: 内存整体优化效果

Figure 4.2: Overall Performance (Memory)

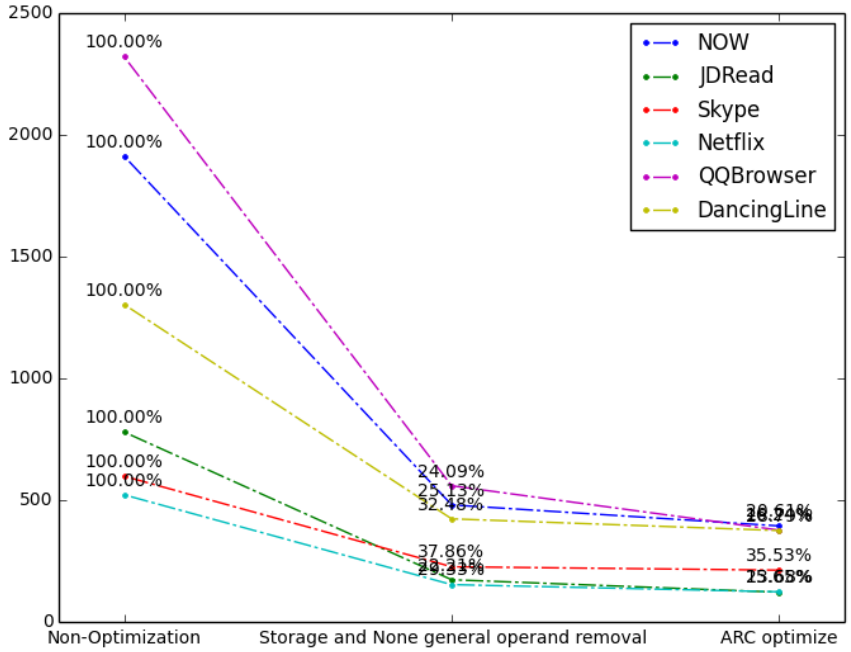


Figure 4.3: 时间整体优化效果

Figure 4.3: Overall Performance (Time)

4.5 本章小结

LLVM IR 是 LLVM 的核心，大部分相关工作都是围绕着 LLVM IR 开展的，如方舟编译器[7]，苹果的Clang等。本章在向 LLVM IR 转化的基础上，开展了相关的优化工作，优化所节省的时间使得分析工具适合进行大规模 iOS 应用分析，优化所节省的内存也为分析 iOS 应用中的库提供了空间。

[7] https://www.openarkcompiler.cn/home

Chapter 5

面向 iOS 应用的指针分析

5.1 相关背景

指针分析是程序分析的基础，可用于程序验证、程序优化、安全分析、程序理解等领域。该分析可提升其他领域，如活跃分析（live-analysis）等的分析精度。在本工作中，指针分析主要用于后续的切片分析。指针分析方法有很多种，包括：流敏感（flow-sensitive）、流不敏感（flow-insensitive）；上下文敏感（context-sensitive）、上下文不敏感（context-insensitive）；域敏感（field-sensitive）、域不敏感（field-insensitive）；基于等价关系（unification-based）、基于子集合关系（subset-based）；过程内（intra-procedural）、过程间（inter-procedural）等分析方法。

与指针分析等价的别名分析[1]工作早就开始，Banning [2] 早在 1979 年就考虑并设计了过程间的别名分析方法。随后的分析工作大多采用传统的数据流分析算法来求解相关问题，如 Emami [6, 7] 采用了标准的 gen-kill 数据流方程来进行指向分析。当前基本的指针分析方法有两种：(i) 1994 年，Andersen [1] 在其博士论文中，将指针指向转化为约束包含，从此确定了 Andersen 风格（inclusion-based、constraint-based）的指针分析；(ii) 1996 年，Steensgaard [32] 将指针指向转化为合并问题，从此确定了 Steensgaard 风格（unification-based、equivalence-based）的指针分析。简单来讲，当一个指向集遇到另外一个指向集时，Steensgaard 指针分析算法将两个指向集合并，而 Andersen 指针分析算法将指向集进行传递。

Shapiro 和 Horwitz [31] 比较了不同的指针分析方法 [1, 30, 32]。在这些方法中，Andersen [1] 指针分析算法最准确，但是算法时间复杂度为 $O(n^3)$。Steensgard 的算

[1]别名分析是用来确定两个指针是否指向内存中的同一个对象，指针分析是用来确定一个指针可能指向内存中的哪些对象。虽然表述不同，其背后的工作机制相同。

法 [32] 时间复杂度为 $O(n)$，但准确率很低。SHAPIRO [30] 等提出的第三种算法在时间和精确度之间作出了平衡。Andersen 和 Steensgard 算法的主要区别是，Andersen 使用了包含关系（inclusion relations）而 Steensgard 基于等价关系（equality relations）构建他们的分析方法。

Andersen 风格的指针分析算法使用了流不敏感的分析方法，因为在准确率和开销之间的平衡 [11] 而在商业产品上广泛使用，如 GCC 和 LLVM 等商业工具使用了 Andersen 风格的指针分析 [23]。相比 Steensgaard 风格指针分析方法，Andersen 风格的指针分析后来得到深入研究，主要原因包括：(i) 相比 Steensgaard 风格的指针分析，Andersen 风格的指针分析精确度更高；(ii) Andersen 风格的指针分析在算法上有很大的提升空间，进而吸引了更多的研究人员。在 Andersen 风格的指针分析领域，比较有代表性的研究工作是 Hardekopf [11] 的指针分析算法。

Andersen 风格的指针分析方法包含两个步骤：(i) 约束生成；(ii) 约束求解。因此，本章结合 iOS 应用 IR 的特性和 Hardekopf [11] 的指针分析算法，描述基本的约束图的生成、求解及优化过程。

5.2 Andersen 指针分析算法及优化

Andersen 在其博士论文 [1] 中设计了一种基于约束的指针分析方法，其通过约束规范来表示实际程序的指向信息，在此基础上，使用相对原始的方式对指针分析方法进行描述。直到 Fähndrich 等人 [8] 通过有向图来表示约束关系，将指向求解过程表示成有向图的传递闭包计算问题，至此，吸引大量研究人员进入到 Andersen 风格指针分析领域。Andersen 风格的指向求解过程分为两个部分，分别为：约束搜集部分和从约束求解指向信息的过程。

5.2.1 约束搜集

Andersen 风格指向分析中的约束系统是从代码中搜集、构建。通过顺序扫描代码中指向相关的操作，如表 5.1的程序代码所示，即可构建三类约束 [12]，分别为基本约束、简单约束和复杂约束。表 5.1中，对于一个变量 v，$pts(v)$ 表示 v 的指向集，$loc(v)$ 表示 v 的抽象内存地址。

- 对于引用操作（$a = \&b$），其含义为 $loc(b) \in pts(a)$，即 b 的地址属于 a 的指向集；
- 对于别名生成操作（$a = b$），其含义为 $pts(a) \supseteq pts(b)$，即 b 的指向集是 a 的指向集的子集，而不是等价关系（Steensgaard 风格的指针分析使用等价关系）；

- 对于解引用读取操作（$a = *b$）和解引用写入操作（$*a = b$），其含义为 $a \supseteq *b$ 和 $*a \supseteq b$。

5.2.2 指向信息计算

为了对进行指向分析，可以构建一个混合了指向关系和包含关系的有向图 $G^* =< N, E^* >$，其中 N 是程序中的变量或抽象的内存位置，E^* 代表指向关系或包含关系的一条边。然后运用传递闭包的计算来将包含关系的边 E^* 转化为指向关系的边 E，最终形成有向图 $G =< N, E >$，其中的 E 表示变量的指向关系。

E^* 的生成过程： 对于基本约束 $loc(b) \in pts(a)$，指向信息处理比较简单，即 $a \to b$，此时 E^* 其代表一种指向关系。对于简单约束指向信息 E^* 的生成方法为：

$$a \to^* b \in E \; iff \; a \subseteq b \tag{5.1}$$

此时 E^* 代表一种指向传递关系；复杂约束（指针解引用）相关的操作，处理过程稍微复杂，通过如下过程将其转化为基本约束：

$$a \to^* b \in E \; iff \; a \in pts(v) \wedge pts(*v) \subseteq pts(b) \tag{5.2}$$

$$a \to^* b \in E \; iff \; pts(a) \subseteq pts(*v) \wedge b \in pts(v) \tag{5.3}$$

E 的生成过程： 此时 G^* 中的边混合指向关系和指向传递关系两种边，针对指向传递关系的边，使用规则：

$$(loc(v) \in pts(a)) \vee (a \to b \in E) \Rightarrow loc(v) \in pts(b) \tag{5.4}$$

即可将其转化为指向关系的边，最终构建的 G 代表指向的关系，通过 G 可以求出某个变量的具体指向。图 5.1是针对代码 5.1来构建 G 的一个过程。

Table 5.1: 约束类型

Table 5.1: Constraint Types

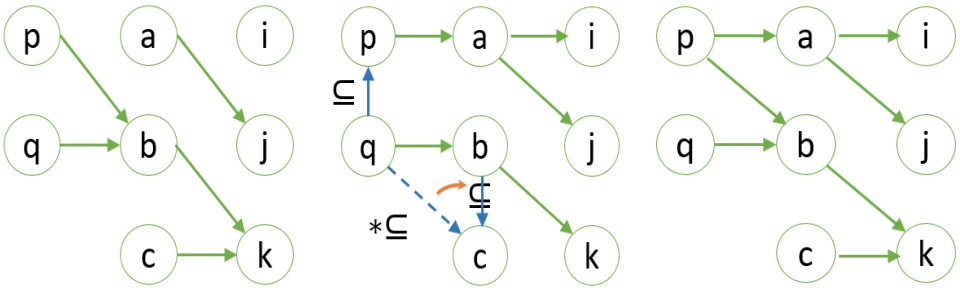
约束类型	程序代码	约束	含义
基本约束（base）	$a = \&b$	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
简单约束（simple）	$a = b$	$a \supseteq b$	$pts(a) \supseteq pts(b)$
复杂约束 1（complex）	$a = *b$	$a \supseteq *b$	$\forall v \in pts(b) : pts(a) \supseteq pts(v)$
复杂约束 2（complex）	$*a = b$	$*a \supseteq b$	$\forall v \in pts(a) : pts(v) \supseteq pts(b)$

Listing 5.1: 包含几种约束类型的 c 代码

```
1  int i, j, k;
2  int *a = &i;
3  int *b = &k;
4  a = &j;
5  int **p = &a;
6  int **q = &b;
7  p = q;
8  int *c = *q;
```

Figure 5.1: 代码5.1的指向分析

Figure 5.1: Point to analysis for code 5.1



(a) 实际的指向关系 **(b)** 混合了约束图和指向图的 G^* **(c)** 指向图 G

图 5.1b中，由于 Andersen 风格的上下文不敏感属性，导致变量 a“安全” 的指向 i 和 j，使得该分析方法在精度上的产生了一些损失。图 5.1b中，q 和 p 的 $\rightarrow^*$ 是由第7行代码所表示的包含关系（$\subseteq$）生成，其指向代表了约束传递关系。运用复杂约束转化规则5.2，可以将 q 和 c 的传递关系转化为 b 和 c 的传递关系。最后使用传递关系和指向关系转化规则5.4，将这两个传递关系转化为指向关系 $p \rightarrow b$ 和 $c \rightarrow k$。

对最终指向信息的计算是基于传递关系的迭代计算，该过程迭代进行，直到系统最终确定指向关系。该计算过程是一个计算图的传递闭包的过程，其算法复杂度为 $O(n^3)$。

5.2.3 指向信息计算算法描述

以上是指向信息计算的形式描述，实际的指向信息计算如算法 2所示。该算法不会实际生成 G^*，而是通过传递闭包计算直接生成 G。算法将 G 中的全部节点 (V) 初始化为 worklist(W)，其中包含传递关系，通过对 worklist(W) 中节点进行迭代分析，最终形成指向关系。指向关系计算过程中，算法对每个从 worklist 中取出的节点 n，做了如下处理：

- 对于每个 $loc(v) \in pts(n)$ 的处理：对于任意的约束 $a \supseteq *n$，增加一条 $v \to a$ 的有向边；对于约束 $n \supseteq *b$，添加边 $b \to v$。每个新增边对应的起点被重新加入到 worklist 中。
- 对于每个出边 $n \to v$，将 $pts(n)$ 传播至节点 v，每个指向集发生改变的节点都被重新加入到 worklist 中。

迭代持续进行，直至 $W = \phi$。该计算过程与图 5.1b到图 5.1c的转化方法一致。

Algorithm 2 动态传递闭包计算

Input: $G =< V, E >$

```
1: W ← V ;
2: while W ≠ φ do
3:     n ← SELECT − FROM(W)
4:     for v ∈ pts(n) do
5:         for constraint a ⊇ *n do
6:             if v → a ∉ E then
7:                 E ← E ∪ {v → a}
8:                 W ← W ∪ {v}
9:             end if
10:        end for
11:        for constraint *n ⊇ b do
12:            if b → v ∉ E then
13:                E ← E ∪ {b → v}
14:                W ← W ∪ {b}
15:            end if
16:        end for
17:    end for
18:    for n → z ∈ E do
19:        pts(z) ← pts(z) ∪ pts(n)
20:        if pts(z) changed then
21:            W ← W ∪ {z}
22:        end if
23:    end for
24: end while
```

5.2.4 Andersen 风格指针分析算法优化

Fähndrich 等人 [8] 将指向问题通过有向图来表示的同时，指出优化对于求解的性能至关重要，后续大量研究人员对 Andersen 风格指针分析进行了优化。

目前在基于包含的指针分析算法的效率改进措施本质上并没有减小指针分析算法的复杂度，而是通过显著地减少 n 来获得性能的提升。因为 Anderson 风格指针分析算法的复杂度为 $O(n^3)$，其中的 n 表示节点的数量，因此减少节点的数量，对于求解速度有很大的提升。如算法 2所示，约束求解以复杂约束为驱动进行迭代，其间不断有新边加入到约束图中，通过检测和合并不断更新的约束图上的强连通分量可以显著地减少约束图上冗余的指向集传递，同时有效地降低约束求解过程中的迭代次数。因为约束图中强连通分量具有相同的指向集，所以约束图中的强连通分量可以合并以降低迭代求解的时间、空间开销。在图算法领域已有一些经典的算法能够高效地检测有向图上的强连通分量，如 Tarjan 算法 [33] 和 Nuutila[25] 的算法，指针分析中很早就使用了这些算法，所以优化算法的难点并不在于如何检测强连通分量，而是如何选择检测的时机以及如何控制整个算法的开销等。

为了减少 n 的数量，后续的工作包括在线强连通分量检测与消除优化，基于等价关系的变量替换离线优化技术。Hardekopf 等 [11, 10] 和陈聪明等 [43] 的工作表明，Hardekopf 的在线优化方法是目前在时间和空间开销的比较均衡的指向计算优化方式。其在线优化算法 [11] 分为两个部分，分别为惰性强连通分量检测（Lazy Cycle Detection，LCD）和混合的强连通分量检测（Hybrid Cycle Detection，HCD）；离线优化 [10] 使用了指针等价（PE）和位置等价（LE）的方法来进行变量替换，以此来降低变量和约束的规模。

LCD 方法： LCD 是一种用于检测强连通分量的启发式方法。在通过约束图的边传播指向信息之前，检查源节点和目标节点是否具有相同的指向集；如果相同，那么使用深度优先搜索来检查可能的强连通分量。该技术之所以称为惰性技术，是因为其未在指向集传播之后进行检测。

LCD 是基于以下假设进行的，即当两个节点为强连通分量时，他们通常具有相同的指向集，否则就不必浪费时间尝试检测强连通分量。LCD 另外的一个改进是，对具有相同指向集但不是强连通的相关节点不再进行检测。LCD 的这些假设导致无法检测所有的强连通分量。

惰性强连通检测算法如算法3第20-23行所示。在将指向集从一个节点传播到另一个节点之前，先检查是否满足两个条件：(i) 两个节点的指向集相同；(ii) 之前没有在此边上进行过搜索。如果满足这两个条件，则触发以目标节点为根节点的强连通分量检测。如果存在一个环路，则将强连通分量合并，否则，会记录该边，以便以后不再

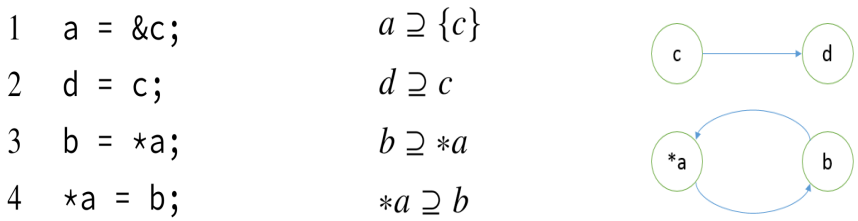
重复尝试。

HCD 方法：强连通分量检测可以在实际指向分析之前离线进行，如使用 Rountev 等人 [29] 提出的离线变量替换方法。但是，强连通分量往往在约束图添加新的边后才出现，因此才需要在线惰性强连通分量检测技术，比如惰性强连通分量检测。但在线强连通分量检测的缺点是，它需要遍历约束图多次以搜索强连通分量，而这些重复的遍历开销非常大。为此，提出了混合循环检测（HCD）技术，该技术结合了离线分析和在线分析来检测强连通分量，从而能检测指针分析过程中在线创建的强连通分量，但无需遍历约束图。

HCD 离线分析是在实际指针分析之前进行的分析，其复杂度为 $O(n)$。HCD 建立了约束图的离线版本，其中每个程序变量都对应一个节点，另外每个解引用的变量也有一个引用节点。每个简单和复杂约束都有一个有向边，如 $a \supseteq b$ 对应边 $b \to a$，$a \supseteq *b$ 对应边 $*b \to a$，$*a \supseteq b$ 对应边 $b \to *a$，约束图忽略基本约束。图5.2说明了此过程，如图所示，$*a$ 和 b 在一个环路中，从此可以推论 b 在 b 和 a 的指向集变量所组成的环路中。

Figure 5.2: HCD 离线分析举例

Figure 5.2: Example of offline HCD

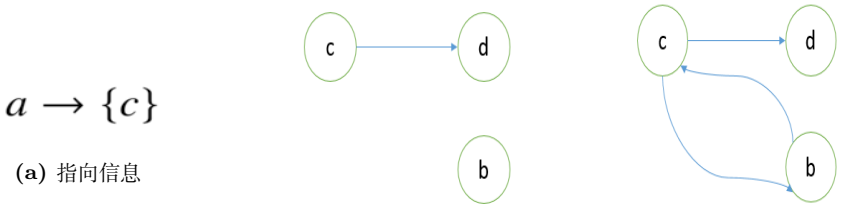


(a) 源程序 **(b)** 约束 **(c)** 离线约束图

一旦建立了约束图，就可以使用 Tarjan 的算法检测强连通分量（SCC）。任何仅包含非解引用节点的 SCC 都可以马上进行合并。对于包含解引用节点的 SCC 则存在一些问题，因为包含解引用节点（例如 $*n$）的 SCC 实际上意味着 n 的指向集是 SCC 的一部分；但由于尚不知道指向集，因此无法合并该 SCC。鉴于离线分析知道哪些变量的指向集将成为 SCC 的一部分，而在线分析（即指针分析）则知道变量的实际指向集，所以使用了混合强连通变量检测。图 5.3显示了离线约束图中的 SCC 包含解引用节点时，是如何影响在线分析的。

Figure 5.3: HCD 在线分析举例

Figure 5.3: Example of online HCD



(a) 指向信息

(b) 加入边之前约束图

(c) 加入边之后约束图

通过在离线约束图中查找由多个节点组成且还包含至少一个解引用节点的 SCC 来完成离线分析。然后对于每个 SCC，先选择一个非引用节点 b，然后对于同一 SCC 中的每个解引用节点 $*a$ 组成元组 (a, b)，并存储在列表 L 中。该元组为在线分析提供信息，表明 a 的指向集在包含 b 的 SCC 中，因此，可以将 a 的指向集可以和 b 进行合并。

在线分析算法见 4。该算法与基本算法 2类似。除了处理节点 n 时，首先检查 L 是否存在形式为 (n, a) 的元组。如果有，则会首先将节点 a 和 n 的指向集合并。

PE 方法：指针等价的优化方法就是查找程序中指向相同的变量集合，在指针等价集的基础上，通过替换变量的方法来降低指针分析的开销。指针等价的方法与 OCD 的检测方法类似，都是尝试查找强连通分量。Hardekopf 等 [29] 提出的离线变量替换（Offline Variable Substitution，OVS）的基础上，进一步提出了基于 Hash 值标号（Hash-based Value Numbering，HVN）算法，来查找更多的指针等价变量。除此外，还提出了 HR 算法和 HU 算法（合称 HRU），其中 HU 算法是基于离线约束图上的抽象指向集的传递来对 HVN 算法进行扩展。

LE 方法：LE 算法将位置作为研究对象，来查找指向相同的变量。即位置等价的变量可以进行替换。其计算过程与 PE 方法类似。

5.2.5 面向 iOS 应用可执行代码的指针分析方法

指针分析主要分析变量 (v)ariable 和位置 (l)ocation 之间的指向关系。在变量 v 在 LLVM IR 中表现为 SSA 中的变量，l 是这些变量的抽象位置。

Table 5.2: 约束类型在 LLVM IR 上的表现

Table 5.2: Relation between constraint and LLVM IR

约束类型	Objective-C 代码	LLVM IR	等价语意
基本约束	int *a, b; a = &b;	[0x1000062E4] %X8_13 = add i64 %SP_init, -12 [0x1000062E8] %1 = add i64 %SP_init, -8 [0x1000062E8] %2 = inttoptr i64 %1 to i64* [0x1000062E8] store i64 %X8_13, i64* %2, align 1	%X8_13 = %SP_init - 12 %1 = %SP_init - 8 %2 = %1 *%2 = %X8_13
简单约束	int *a1, *b1; a1 = b1;	[0x1000062EC] %3 = add i64 %SP_init, -32 [0x1000062EC] %4 = inttoptr i64 %3 to i64* [0x1000062EC] %X8_14 = load i64, i64* %4, align 1 [0x1000062F0] %5 = add i64 %SP_init, -24 [0x1000062F0] %6 = inttoptr i64 %5 to i64* [0x1000062F0] store i64 %X8_14, i64* %6	%3 = %SP_init - 32 %4 = %3 %X8_14 = *%4 %5 = %SP_init - 24 %6 = %5 *%6 = %X8_14
复杂约束 1	int a2, *b2; a2 = *b2;	[0x1000062F4] %7 = add i64 %SP_init, -48 [0xFFFFFFFFF] %8 = inttoptr i64 %7 to i32** [0xFFFFFFFFF] %X8_151 = load i32*, i32** %8, align 1 [0x1000062F8] %W9_4 = load i32, i32* %X8_151, align 1 [0x1000062FC] %9 = add i64 %SP_init, -36 [0x1000062FC] %10 = inttoptr i64 %9 to i32* [0x1000062FC] store i32 %W9_4, i32* %10, align 1	%7 = %SP_init - 48 %8 = %7 %X8_151 = *%8 %W9_4 = *%X8_151 %9 = %SP_init - 36 %10 = %9 *%10 = %W9_4
复杂约束 2	int *a3, b3; * a3 = b3;	[0x100006300] %11 = add i64 %SP_init, -60 [0x100006300] %12 = inttoptr i64 %11 to i32* [0x100006300] %W9_5 = load i32, i32* %12, align 1 [0x100006300] %X9_6 = zext i32 %W9_5 to i64 [0x100006304] %13 = add i64 %SP_init, -56 [0x100006304] %14 = inttoptr i64 %13 to i64* [0x100006304] %X8_16 = load i64, i64* %14, align 1 [0x100006308] %15 = inttoptr i64 %X8_16 to i32* [0x100006308] store i32 %W9_5, i32* %15, align 1	%11 = %SP_init - 60 %12 = %11 %W9_5 = *%12 %X9_6 = %W9_5 %13 = %SP_init - 56 %14 = %13 %X8_16 = *%14 %15 = %X8_16 *%15 = %W9_5

5.2.5.1 建立面向 iOS 应用可执行代码的约束系统

Andersen 风格指针分析算法通过扫描代码建立约束系统，表 5.2列出了常见约束和 LLVM IR 的对应关系。在处理指向关系时，需要建立全局变量指向关系、函数实参和形参指向关系、函数或函数指针、与类层次结构相关的函数（OC 语言的协议）指向关系、调用方和被调用方返回值的指向关系等以建立约束。

在建立这些关系时，需要针对二进制代码做额外的一些处理。在 iOS 应用编译过程中，LLVM 的后端从 LLVM IR 中去除了函数原型信息，没有这些信息，将无法建立过程间的指向关系。编译过程去除了本地变量信息，没有这些信息，指向关系将不完整。为了恢复这些信息，通过使用调用约定[2]来推测参数和返回信息。

[2] https://developer.arm.com/docs/ihi0055/latest/procedure-call-standard-for-the-arm-64-bit-architecture

局部变量信息恢复：函数的局部变量被编译成栈上的位置 l，源代码中针对变量的访问被转化为通过栈桢寄存器加静态偏移来访问栈。针对栈 l 的访问除用栈桢寄存器（SP）加静态偏移（*add* 或者 *sub* 等操作）外，还可以通过栈桢的赋值操作，用其他寄存器加静态偏移来访问。因此可以产生多个 v 指向同一个 l，而这些 v 并不是通过简单约束对应的操作产生，如果不考虑这些额外的关系，则生成的约束关系不完整。

LLVM IR 采用了 SSA 表示，指向信息的赋值、计算都将形成新的变量 v，需要检测出指向同一位置（栈地址）的不同变量来进行指向分析。栈的指向信息计算过程如算法 5所示，算法从方法中扫描 *inttoptr* 和 *store* 指令，把符合特定模式的操作数对应的生成指令加入工作表中（2-9行）。接下来，沿着定值-使用链确定工作表中变量的确切指向（10-30行），该过程中，会对栈偏移的算数运算进行模拟执行（15-18行），对 ϕ 节点（19-21行）将沿所有可能定值节点继续沿着定值-引用链遍历，遍历在遇到针对上下文的寄存器的访问时终止（22-27行）。随后，将当前变量可能的指向进行搜集整理。通过这种方式，可以确定 SSA 变量的基本指向关系，如代码 5.2所示。代码中，对当前函数来讲，v（SSA 变量）：%11，%17，%23 指向同一个 l：-40（栈中的抽象位置）。

Listing 5.2: 栈上的基本指向关系

```
-40                                                    1
  %11 = add i64 %SP_init                               2
  %17 = add i64 %SP_32, 24                             3
  %23 = add i64 %SP_32, 24                             4
-32                                                    5
  %9 = add i64 %SP_init, -32                           6
  %21 = add i64 %FP_26, -16                            7
-24                                                    8
  %7 = add i64 %SP_init, -24                           9
  %19 = add i64 %FP_26, -8                             10
-20                                                    11
  %5 = add i64 %SP_init, -20                           12
  %25 = add i64 %FP_26, -4                             13
  %27 = add i64 %FP_26, -4                             14
```

参数和返回信息恢复：为建立实参和形参之间的指向关系，函数返回值和调用方接收值之间的关系，需要对函数参数，返回值的信息进行恢复。

编译后的代码将函数的参数通过 'X0-X7' 寄存器传递，当参数数量足够多时，部分参数保存在栈上，并使用 'FP' 寄存器来寻址。因此，参数的识别包含两部分内容，即寄存器参数识别和栈参数识别。这些参数在使用前，一般不会存在赋值操作，因此使用模式：read-before-write 来识别这些参数。该模式的识别在具体实现上，通过查找基本块中对上下文的寄存器的第一条指令是否为 *load* 指令来完成寄存器参数的确认；对于栈参数，借助局部变量指向关系搜集的计算结果和支配树的计算结果来对栈

上的参数进行识别。参数信息的恢复过程如算法 6所示，其中1-12行为寄存器参数识别方法，其余为栈参数识别方法。通过该方法，可匹配如代码 5.3所示模式。

Listing 5.3: 寄存器参数举例

```
[0x100005F7C] %X8_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 13
[0x100005F7C] %X8_init = load i64, i64* %X8_ptr, align 4
```

AArch64 调用约定中指定 'X0' 寄存器用于存储返回值，因此对某个指定的函数，通过访问每个基本块，并判断基本块的是否以 *ret* 结束，如果是，则该基本块为一个函数出口。随后，在基本块中逆向查找针对 'X0' 或其别名的存储操作，以此定位函数的返回信息。该过程如算法7所示。通过该方法，可匹配代码 5.4所示模式。

Listing 5.4: 返回值识别举例

```
[0x100005F8C] %X0_ptr = getelementptr inbounds %regset, %regset* %0, i64 0, i32 5
[0xFFFFFFFFF] store i64 %X0_3, i64* %X0_ptr
```

扫描代码建立约束系统：在以上恢复的信息的基础上，对 iOS 应用的 LLVM IR 进行完整扫描，以建立约束系统。

Andersen 的方法是基于 C 语言的，因此，有些 LLVM IR 的关键特征并没有覆盖到。iOS 应用的 LLVM IR 中所有的内存访问是通过 *inttoptr* 指令将一个整形转化为指针来实现指向。如果不考虑这个指令特征，指向集中所有指针将指向抽象地址 *Unknown*。基于 Hardekopf 和 Lin[11] 的约束生成方法，修改约束生成规则来覆盖 *inttoptr* 指令，识别出 LLVM IR 中 *inttoptr* 指令的操作数来建立指向信息。

具体的修改如表 5.3所示，LLVM IR 中的指向关系并不如源代码中那么明显，需要分析指令的操作数的实际特征以建立指向关系。部分 LLVM IR 中的指令和约束的关系如表 5.4所示。对于其中有些指令，如 'Instruction::Store'，需要结合栈分析的结果，将别名关系加入到约束系统中。对需要根据操作数类型生成约束的指令，如 'Instruction::Load'，'Instruction::Store' 指令，考虑 ϕ 节点。针对 'NSDictionary'，'NSArray' 等复杂对象，在针对 'objcMsgSend' 的处理过程识别出 'Receiver' 后，使用域敏感（field-sensitive）的处理方法来建立指向关系。

过程内与过程间指针分析： Andersen[1] 在进行指针分析时，通过不建立调用方和被调用方参数指向、返回值和返回值接收者之间的指向来将过程间指针分析转化为过程内分析。与 Andersen 不同，我们通过只扫描包含切片准则的函数内的指令，来将过程间分析转化为过程内分析。通过这种方式，只需要扫描指定函数，而无需为其他

Table 5.3: LLVM IR 指令和约束的关系

Table 5.3: LLVM IR Instruction and Constraint

LLVM IR 指令	生成的约束类型	解释
Instruction::Alloca	基本约束	N/A
Instruction::Invoke	简单约束	建立形参和实参之间的指向关系
Instruction::Ret	简单约束	建立返回值和调用方接收值之间的指向关系
Instruction::Load	基本约束	Load 的操作数为堆对象
	简单约束	Load 的操作数为未初始化对象
	复杂约束 1	Load 的操作数为已初始化对象
Instruction::Store	基本约束	Store 的操作数为立即数
	复杂约束 2	Store 的操作数为指针对象
Instruction::PHI	简单约束	建立 ϕ 和所有可能值之间的指向关系
Instruction::Add	简单约束	操作数为地址类型

函数建立约束系统，在最终效果上，与 Andersen 的方法结果一致。而在系统开销上，由于不采集无关的约束，相比 Andersen 的方法，其开销更小。

5.2.6 约束求解与优化

约束的求解与优化采用了标准的方法。在指向关系的存储方面，使用了二元决策图（Binary Decision Diagram，BDD）[3] 的方式来存储，以降低位向量方式（bit vector）的空间开销。

约束求解：约束求解使用如 2所示的标准算法。指针分析模块实现了过程内和过程间指针分析。对于过程间分析，需要识别函数的调用目标以建立调用方和被调用方的指向关系（参数、返回值）。因为 iOS 应用是面向运行时的语言（runtime-oriented）Objective-C 和 Swift 编译而来。LLVM 前端将直接的方法调用改写为动态分派调用。*objc_msgSend()* 函数是 Objective-C 运行期库，该函数在运行期确定了调用那个方法。该函数接收分别指定一个类或对象的参数和一个方法参数，这两个参数分别对应 ARMv8 中的 'X0' 和 'X1' 寄存器。这两个寄存器的内容为指向二进制文件中实际数据的指针。因此，识别这两个寄存器的指向信息可以协助过程间指向信息的构建。基于此，通过为在调用 *objc_msgSend()* 时实际引用的函数加入边，来恢复正确的指向关系与调用图。其他 Objective-C 特有的属性也需要处理，比如 Blocks 和 Fast Enumeration。对于这两个属性，约束生成也需要做适配，使其能识别访问栈地址的指令。

Andersen [1] 描述了为每条指令生成约束的算法。该算法在一条指令的所有值都确定的情况下有效。在处理函数指针或者调用 Objective-C 函数时，指向信息必须提前知晓。因此，我们使用迭代的方式来使约束生成过程适配能处理 call 指令以及其

所需的指针信息。

约束优化：在将 Andersen 的初始算法运用到 iOS 应用时，iOS 应用的规模往往会导致其性能急剧降低。为了解决这个问题，后续有大量工作 [3, 10, 11, 12, 28] 对其进行改进。其中，Hardekopf 和 Lin[11] 将时间复杂度基本降低为线性的同时，提供了和 Andersen 算法几乎一致的结果。

在优化方面，实现了离线优化的 HVN，HU 算法，在线优化的 LCD，HCD 算法。鉴于离线优化算法取决于前驱节点信息，因此在优化之前，识别每个节点的前驱节点。

5.3 指向分析模块开销

在约束求解过程中，函数调用的指向问题被解决，因此针对函数调用会有新的约束信息不断加入到约束图中。因此，求解过程会持续进行，直到不再产生新的约束（所有调用目标都已经解决或无法解决）。由表 5.4可见，过程间的指针分析由于函数调用的指向信息不断加入到约束系统，而指向分析的复杂度是 $O(n^3)$，因此其求解过程非常耗时。这里只使用了针对 GCDWebServer 的切片准则测试了两个应用，其他应用由于未使用这个函数，导致过程内指针分析提前退出，无法进行测试比较。

Table 5.4: 指向分析方法开销

Table 5.4: Overhead of point to analysis

	过程间分析			过程内分析	
	函数数量	约束数量[a]	时间开销（S）[b]	约束数量[a]	时间开销（S）[c]
NOW	172,942	19,787,623	56427.7490	5,447	15.0482
京东读书	68,376	8,490,563	Killed after 2 days running	3,147	11.7077
均值	120,659	14,139,093	N/A	4,297	13.37795

[a] 这里的约束数量指约束系统初始约束的数量；

[b] 使用了全部的优化算法，包括 HVN，HU，LCD，HCD。过程间分析由于内存耗尽额外引入内存压缩、置换等时间，如分析 NOW 耗时约 15.67 小时，但实际运行超过了 1 天。

[c] 使用了部分的优化算法，包括 LCD，HCD。

由表可见，采用过程间分析，即使引入了所有的优化手段，指针分析的时间开销仍然不可接受。如分析 NOW 耗费了 15.67 小时，而分析京东读书时，因为耗时而

被系统杀死。同样的应用采用过程内的指针分析，指针分析通常在分钟级别内即可完成。

5.4 按需进行的约束搜集

由表 5.4可见，对分析目标进行过程间的指向分析在时间上的开销是不可接受的，而过程内的指向分析在精度上又存在一定的损失。通过对分析时间和分析精度的平衡，我们提出了一种按需进行的指针分析方法，该方法对包含切片准则的函数进行指向分析，在识别出调用目标后，对调用目标函数继续进行指向分析，直至没有新的约束加入。在这种策略下，指向分析和程序切片是交替进行的。

5.5 本章小结

指针分析为切片准备了准确的指向信息。我们针对 LLVM IR 和 Objective-C 的特点来生成上下文、流不敏感的约束。

指针分析又称为别名分析。在编译领域对别名关系的使用是保守的，即如果能证明 p 绝对（must）是 q 的一个别名，该别名关系才可用于编译优化。程序分析领域则采用了激进的策略，即如果能证明 p 可能（may）是 q 的一个别名，该关系即可用于程序分析。

Andersen 为其算法定义了上下文敏感(context-sensitive)和上下文不敏感(context-insensitive）两个版本，虽然上下文敏感版本为不同路径分别保存了信息，因此精度较高，但其时间复杂度为指数级 [7, 37]，因此，上下文敏感的指针分析不适用于包含大量调用上下文的大型应用。我们更倾向使用上下文不敏感的方法来计算指向集。

流敏感（flow-sensitive）指针分析考虑程序的控制流，并且针对每条指令计算指向集。流不敏感（flow-insensitive）指针分析不关注程序中的分支，并且将所有的信息搜集在一个指向集中。Hind 和 Pioli[13] 证明了流敏感的指针分析相对于流不敏感的指针分析在精度上并没有多大的提升，因此，我们倾向使用流不敏感的指针分析方法。我们使用流不敏感的分析方法是基于 SSA IR 表示的，SSA 中的每个赋值操作都对应一个新的变量。在该特性下，p 的指向并不会被传递到之前的语句。而这一特点刚好是 Holst[14] 提出的优化方法 “poor man”，且 SSA 表示隐含了控制信息，因此，在本质上，我们的分析方法属于部分流敏感分析方法。

综合了分析精度和分析时间上的开销，在面向 iOS 应用程序的指针分析实践中，我们近一步采用了按需进行的过程间指针分析方法。该方法在精度上会产生一定的损失，但可以运用在大规模 iOS 应用的测量上。

Algorithm 3 惰性强连通分量检测

Input: $G =< V, E >$;

$R \leftarrow \phi$
$W \leftarrow V$
while $W \neq \phi$ **do**
    $n \leftarrow SELECT - FROM(W)$
    for $v \in pts(n)$ **do**
        for constraint $a \supseteq *n$ **do**
            if $v \rightarrow a \notin E$ **then**
                $E \leftarrow E \cup \{v \rightarrow a\}$
                $W \leftarrow W \cup \{v\}$
            end if
        end for
        for constraint $*n \supseteq b$ **do**
            if $b \rightarrow v \notin E$ **then**
                $E \leftarrow E \cup \{b \rightarrow v\}$
                $W \leftarrow W \cup \{b\}$
            end if
        end for
    end for
    for $n \rightarrow z \in E$ **do**
        if $pts(z) = pts(n) \wedge n \rightarrow z \notin R$ **then**
            DETECT-AND-COLLAPSE-CYCLES(z)
            $R \leftarrow R \cup \{n \rightarrow z\}$
        end if
        $pts(z) \leftarrow pts(z) \cup pts(n)$
        if $pts(z)$ changed **then**
            $W \leftarrow W \cup \{z\}$
        end if
    end for
end while

Algorithm 4 混合的强连通分量检测

Input: $G =< V, E >$

1: $W \leftarrow V$;
2: **while** $W \neq \phi$ **do**
3: $n \leftarrow SELECT - FROM(W)$
4: **if** $(n, a) \in L$ **then**
5: **for** $v \in pts(n)$ **do**
6: $COLLAPSE(v, a)$
7: $W \leftarrow W \cup \{a\}$
8: **end for**
9: **end if**
10: **for** $v \in pts(n)$ **do**
11: **for** constraint $a \supseteq *n$ **do**
12: **if** $v \rightarrow a \notin E$ **then**
13: $E \leftarrow E \cup \{v \rightarrow a\}$
14: $W \leftarrow W \cup \{v\}$
15: **end if**
16: **end for**
17: **for** constraint $*n \supseteq b$ **do**
18: **if** $b \rightarrow v \notin E$ **then**
19: $E \leftarrow E \cup \{b \rightarrow v\}$
20: $W \leftarrow W \cup \{b\}$
21: **end if**
22: **end for**
23: **end for**
24: **for** $n \rightarrow z \in E$ **do**
25: $pts(z) \leftarrow pts(z) \cup pts(n)$
26: **if** $pts(z)$ changed **then**
27: $W \leftarrow W \cup \{z\}$
28: **end if**
29: **end for**
30: **end while**

Algorithm 5 栈指向信息恢复

Input: Func

Output: StackAccessModel: <Location, Insts>

```
StackAccessInstructions ← ϕ
for Inst ∈ Func.InstList do
    if Inst.opcode = Instruction::IntToPtr then
        StackAccessInstructions.add(Inst.getOperand(0).getDefInst())
    end if
    if (Inst.opcode = Instruction::Store) and (Inst.getOperand(0).pattern(value, int)) then
        StackAccessInstructions.add(Inst.getOperand(0).getDefInst())
    end if
end for
for Inst ∈ StackAccessInstructions do
    CurInsts ← Inst
    CurInstLocation ← ϕ
    while True do
        for CurInst ∈ CurInsts do
            if CurInst.opcode = Instruction::Algo then
                CurInstLocation ← CurInstLocation <CurInst.opcode> CurInst.getOperand(1)
                CurInst ← CurInst.getOperand(0).getDefInst()
            end if
            if CurInst.opcode = Instruction::PHI then
                CurInsts ← CurInst.getOperand(0).all().getDefInst()
            end if
            if CurInst.opcode = Instruction::LOAD then
                if CurInst.getOperand(0) = StackPointer then
                    StackAccessModel[CurInstLocation] ← Inst
                    Break
                end if
            end if
        end for
    end while
end for
```

Algorithm 6 函数参数信息恢复

Input: Func, StackAccessModel: <Location, Insts>, DominatorTree(Func)

Output: ParameterRegister, ParameterStack

```
1: for Inst ∈ Func.InstList do
2:     if Inst.opcode = Instruction::GetElementPtr then
3:         if Inst.getOperand(2) = ParameterRegister then // 5 ≤ ParameterRegister ≤ 13
4:             BB ← BBContainer(Inst)
5:             for InstAccess ∈ BB.SubsequenceInst(Inst) do
6:                 if InstAccess.Opcode = Instruction::Load and InstAccess.getOperand(0).getDefInst() = Inst then
7:                     ParameterRegister ← Inst
8:                 end if
9:             end for
10:        end if
11:    end if
12: end for
13: for Location ∈ StackAccessModel.Location do
14:    for Inst ∈ StackAccessModel[Location] do
15:        (LoadInsts ← Inst.getUse()) ⟺ Inst.getUse().opcode = Instruction::Load
16:        (StoreInsts ← Inst.getUse()) ⟺ Inst.getUse().opcode = Instruction::Store
17:    end for
18:    for LoadInst ∈ LoadInsts do
19:        ParameterStack ← LoadInst ⟺ LoadInst dominates StoreInsts
20:    end for
21: end for
```

Algorithm 7 返回值信息恢复

Input: Func

Output: RetAccessModel: <ReturnRegister, Insts>

```
1: for BB ∈ Func.BBList do
2:     if BB.Terminator.Opcode = Instruction::Ret then
3:         for Inst ∈ BB.InstList.ReverseOrder do
4:             if Inst.Opcode = Instruction::Store and Inst.getOperand(1).getDefInst().Opcode = Instruction::GetElementPtr and Inst.getOperand(1).getDefInst().getOperand(2)=ReturnRegister then // ReturnRegister = 5
5:                 RetAccessModel[ReturnRegister] ← Inst
6:             end if
7:         end for
8:     end if
9: end for
```

Chapter 6

面向 iOS 应用的静态切片

程序切片是一种对程序进行自动分解的方法。程序切片以精简的形式来表示程序行为，这个精简的程序又被称为切片。切片在特定行为上，需要忠实的反应原程序。程序切片技术可以运用在调试、并行处理等方面。目前，查找最小切片还是个难题。

静态切片用来确定所有可能影响到执行路径中某个位置的值（切片准则）。切片的结果中包含所有可能路径并且支持对程序的功能得出结论。

6.1 相关背景

程序切片概念于 1979 年由 Mark Weiser 在他的博士论文 [35] 中首次提出，后来又在 1981 年 [36] 以及 1984 年 [24] 进一步阐述了程序切片的思想和实现算法。Weiser 通过对源程序的分析以及对控制流图（Control Flow Graph，CFG）的研究发现：程序的某一个输出只与源程序中部分语句和谓词有关，删除其它的语句和谓词并不影响该输出的结果。这表明，对该输出来说，源程序和删除无关语句和谓词以后所得的可执行程序在语义上是一致的。Weiser 等人把这种只与某个输出有关的语句和谓词构成的程序称为源程序的一种静态切片，并建立了基于 CFG 的计算程序切片的算法。Ottenstein 等人 [27] 分别于 1984 年和 1987 年引入基于程序依赖图（Program Dependence Graph，PDG）的图可达性算法，以便能够计算当时流行的过程内后向切片（Intraprocedural Backword Slicing）。Horwitz 等分别于 1988 年，1990 年和 1992 年引入了前向切片概念及算法；过程间切片概念以及基于系统依赖图（System Dependence Graph，SDG）的两阶段图形可达性算法 [15]。Korel 等于 1988 年和 1990 年引入动态切片的概念和算法 [17, 18, 16, 19, 20]。随着面向象程序设计

语言和设计方法的发展，研究人员把程序切片技术应用到这一领域，于是自 1994 年开始了面向对象的程序切片技术的研究。在这期间，提出了各种面向对象程序切片概念以及解决问题的方法。例如，Chen[4] 的行为切片和状态切片，赵建军 [38, 39] 的动态面向对象程序切片，Larsen[21] 的对象切片，Frank[34] 的 C++ 类层次切片等为面向对象程序切片技术的研究奠定了基础。李必信 [40, 41] 提出了分层切片模型。在本章中，将简要介绍程序切片的概念、介绍 Weiser 程序切片算法以及面向 iOS 应用的静态切片技术实现。

6.2 Weiser 程序切片算法

Weiser 提出的程序切片技术是一种基于数据流和控制流的切片方法。在其论文中 [36]，Weiser 形式化的定义了切片的概念及属性，并给出了切片算法。

6.2.1 Weiser 程序切片相关定义

Weiser 首先定义了有向图、流图、基于状态轨迹的计算和保存相关状态轨迹映射的切片等概念。

有向图（digraph）： 有向图可以用 $< N, E >$ 表示，其中 N 是节点集，E 是包含于 $N \times N$ 中的边集。如果 $(n, m) \in E$，那么 n 是 m 的一个直接前驱（Immediate Predecessor），且 m 是 n 的一个直接后继（Immediate Successor）。从 n 到 m 的一条长度为 k 的路径是相关节点 p_0，p_1 ...，p_k 组成，其中 $p_0 = n, p_k = m$，并且对于所有的 $\{i | 1 \leq i \leq k-1\} \rightarrow (p_i, p_{i+1}) \in E$。

流图（flowgraph）： 流图可以用 $< N, E, n_0 >$ 表示，其中 $< N, E >$ 是一个有向图。n_0 称为初始节点，$n_0 \in N$ 且存在从 n_0 到 N 中任意其他节点的一条路径。如果 m 在从 n_0 到 n 的每条路径上，则称 m 支配（dominator）n。

吊床图（hammock graph）： 吊床图可以用 $< N, E, n_0, n_e >$ 表示，其中 $< N, E, n_0 >$ 和 $< N, E^{-1}, n_e >$ 均为流图。如果 m 和 n 是 N 中的两个节点，m 在从 n 到 n_e 的每条路径上，则称 m 反向支配（inverse dominator）n。

定值/使用： 令 V 是程序 P 的变量集，对 P 中的任何语句 n 都存在如下两个集合：$REF(n) \subseteq V$，$REF(n)$ 是值在 n 处使用的变量。$DEF(n) \subseteq V$，$DEF(n)$ 是值在 n 处改变（定值）的变量。

状态轨迹（state trajectory）： 状态轨迹是一个程序的执行轨迹，该轨迹是在每个语句执行前对所有变量值所做的一个快照。一个程序 P 的长度为 k 的状态轨迹是一个有限的有序对 $(n_1, s_1)(n_2, s_2)...(n_k, s_k)$，其中 $n \in N$，s 是从 V 中变量到值的一个

映射函数。切片是原程序行为的一个投射，该投射必须是某些变量在某些语句的位置的可见值。

切片准则（slicing criterion）： 程序 P 的切片准则是一个元组 $< i, V >$，其中 i 是 P 中的一个语句，V 是程序 P 中变量的子集。一个切片准则 $C =< i, V >$ 确定了一个投射函数 $Proj_C$，除了从 i 开始的状态轨迹，该投射函数去除了状态轨迹中的其他有序对。在剩下的有序对中，除了 V 中变量的值被保留，其他有序对被抛弃。

投射函数： 令 $T = (t_1, t_2 ... t_n)$ 是一个状态轨迹，n 是 N 中任何一个节点，s 是从变量名称到值的一个函数，那么：

$$Proj'_{<i,V>}((n,s)) = \begin{matrix} \lambda & n \neq i \\ (n, s|V) & n = i \end{matrix} \tag{6.1}$$

其中，$s|V$ 是指限定在 V 中的 s，λ 是空字符串。至此，$Proj'$ 可以扩展至整个轨迹，即：$Proj_{<i,V>}(T) = Proj'_{<i,V>}(t_1)...Proj'_{<i,V>}(t_n)$。

切片： 切片是一个程序行为的子集，该子集保存了特定行为的投射。一个程序 P 针对切片准则 $C =< i, V >$ 的切片 S 是一个可执行程序，其具备如下两个特性：

- S 可以从 P 中删除 0 条或者多条语得到。
- 只要程序 P 在输入 I，状态轨迹 T 的情况下停机，那么 S 也会在输入 I，状态轨迹 T' 的情况下停机。并且 $Proj_C(T) = Proj_{C'}(T')$，其中 $C' =< succ(i), V >$，$succ(i)$ 是包含在切片中，原程序或者切片中 i 的最近后继（Nearest Successor）。

6.2.2 切片算法

查找最小切片和停机问题一样是不可解的问题，Weiser 切片算法是基于数据依赖和控制依赖进行的。通常情况下，程序中的每一条语句都会包含一些变量，其值可能影响切片准则中对应变量的值。比如代码 6.1中，第1行之前变量 X 的值将影响第2行之后 Z 的值。

Listing 6.1: 直接相关

```
1   Y = X;
2   Z = Y;
```

使用 R^0_C 表示类似 X 对语句 n 的切片直接相关的变量。上角标 0 表示间接相关的程度。令 $C =< i, V >$ 是一个切片准则，那么 $R^0_C(n)$ 包含所有满足如下条件的变量 v：

$$\begin{aligned} & 1. n = i \wedge v \in V \\ or\quad & 2. n \in immediate\_predecessor(m) \\ & \qquad a) v \in REF(n) \wedge (\exists w)(w \in DEF(n) \wedge w \in R^0_C(m) \\ & \qquad or \\ & \qquad b) v \notin DEF(n) \wedge v \in R^0_C(m) \end{aligned}$$

其中，(1) 是基本情况；(2a) 表示如果 w 是 n 节点前驱节点的相关变量，并且在节点 n，w 被定值，那么节点 w 不再相关，而 w 的定值变量相关；(2b) 表示如果前驱节点的相关变量在节点 n 未被定值，那么在节点 n 仍然相关。令 S_C^0 表示切片中包含的语句，$S_C^0 = \{n | R_C^0(n+1) \cap DEF(n) \neq \phi\}$，其中 R_C^0 是语句到变量集的映射，S_C^0 是语句集合。

$R_C^0(n)$ 的计算过程是数据流方程计算的一个实例，这个过程不足以表示将所有语句包含在切片中。如代码 6.2所示，第1行代码对第5行的 Z 有影响，但是1没有包含在 $S^0_{<5,\{z\}>}$ 中。一般来说，任何能决定 S_C^0 中语句执行的分支语句也应该包含在切片中。

Listing 6.2: 条转语句

```
1  if X < 1
2     Z = 1;
3  else
4     Z = 2;
5  printf(Z)
```

为了解决这个问题，定义 $INFL(B)$ 为一个语句集，该集是从 b 到其最近反向支配点 (nearest inverse dominator) d 的路径 P 上的语句。除非 b 有多个直接后继，否则 $INFL(B)$ 为空。可以使用 $INFL$ 来定义与切片间接相关的分支语句。

$$B_C^0 = \bigcup_{n \in S_C^0} INFL(n) \tag{6.2}$$

为包含所有的间接影响，直接影响 B_C^0 的语句也应该包含进来，然后继续包含影响这些新语句的分支语句。多级影响的完整定义如下：

$$R_C^{i+1}(n) = R_C^i(n) \bigcup_{b \in B_C^i} R^0_{BC(b)}(n) \tag{6.3}$$

$$B_C^{i+1} = \bigcup_{n \in S_C^{i+1}} INFL(n) \tag{6.4}$$

$$S_C^{i+1} = \{n | n \in B_C^i \bigvee (R_C^{i+1}(n+1) \bigcap DEF(n) \neq \phi)\} \tag{6.5}$$

此处的 $BC(b)$ 表示分支语句准则，定义为 $< b, REF(b) >$。R_C^i 和 S_C^i 定义为针对固定 n 和 C 的 i 的函数，它定义了不递减的子集，并且子集上界分别为程序变量集和程序语句集。因此，每个都有一个最小不动点，分别用 R_C 和 S_C 表示。S_C 和 R_C 具有以下特性：

$$S_{<i,A>} \bigcup S_{<i,B>} = S_{<i,A \cup B>} \tag{6.6}$$

$$R_{<i,A>} \bigcup R_{<i,B>} = R_{<i,A \cup B>} \tag{6.7}$$

6.2.3 过程间切片

针对一个调用其他过程或被其他过程调用的过程进行的切片需要保存在调用或者被调用的过程中的语句。跨过程的切片包括两步：

1. 针对过程 P 的单一切片。使用了关于调用其他过程的数据流信息摘要，但没有对其他过程进行切片。
2. 生成针对 P 的调用的过程或者被 P 调用的过程的切片准则。

第一步和第二步在新的切片准则下迭代进行，直到不再生成新的切片准则。

过程间切片最困难的是将通过 R_C 计算的变量集传递到新的过程。假设过程 P 被切片，且过程 P 中有一个位于语句 i 的调用，调用过程 Q，将切片扩展到 Q 的准则为：

$$< n_e^Q, ROUT(i)_{F\to A} \bigcap SCOPE_Q > \tag{6.8}$$

其中，n_e^Q 是 Q 的最后一条语句，$F \to A$ 意味着用实参替换形参，$SCOPE_Q$ 是在 Q 的范围内可访问的变量集，且 $ROUT(i) = \bigcup_{j\in Succ(i)} R_C(j)$。

类似的，假设过程 P 被切片，且过程 P 被过程 Q 的语句 i 调用，则新的准则为：

$$< i, R_C(f_p)_{A\to F} \bigcap SCOPE_Q > \tag{6.9}$$

其中 f_p 是 P 中的第一条语句，$A \to F$ 意味着使用形参来替换实参。

对于每个针对 P 的准则 C，都会生成调用 P 的准则集 $UP_0(C)$ 和 P 调用的准则集 $DOWN_0(C)$，多个 $UP_0(C)$ 和 $DOWN_0(C)$ 合并后得到：

$$UP(CC) = \bigcup_{C\in CC} UP_0(C) \tag{6.10}$$

$$DOWN(CC) = \bigcup_{C\in CC} DOWN_0(C) \tag{6.11}$$

6.2.4 分割编译

分割编译导致的结果是外部调用过程代码不可见，在这种情况下，必须作出最坏假设。即必须假设引用或者定值全部的外部变量。最坏假设确保了切片最大化。

令 ENT_0 是在最坏假设下的准则映射函数。除非 C 是一个入口过程 P 的准则，其他情况下，$ENT_0(C)$ 都为空。ENT_0 的计算过程：

$$ENT_0(C) = (\forall E \in EE)\{< n_e^E, R_C((i) \cup OUT \cup F^E\} \tag{6.12}$$

其中 n_0 是 P 的唯一的初始语句；EE 是所有的入口过程的集合；OUT 是所有外部变量集合；对于每个 $E \in EE$，n_e^E 是 E 的唯一最后一条语句，F^E 是 E 的引用参数集。与 UP 和 $DOWN$ 类似，ENT_0 也可以合并成 ENT。

6.3 面向 iOS 应用的程序切片

面向 iOS 应用的程序切片结合了 Weiser 的切片算法、Objective-C 运行时特点和 ARMv8 的特性计算所有影响切片准则的代码。在基本的切片方法上，遵循了Weiser 提出的方法，如 $R_C^{i+1}(n)$、B_C^{i+1}、S_C^{i+1} 的计算等。当然，也处理了与 Weiser 算法无法适用的场景。

6.3.1 恢复寄存器参数信息

ARMv8 中，参数既可以通过栈传递，又可以通过寄存器传递。Weiser 的方法无法直接应用到反汇编的 LLVM IR 代码上，因为 Weiser 的过程间切片算法是基于所有形参和实参可见的情况下进行的（6.2.3）。指针分析过程中，已经对函数参数和返回值进行了恢复。但是此项工作是针对被调用方的形参的恢复。为了进行切片分析，需要确定调用方的实参传递信息。LLVM IR 中每个过程只有一个形参来代表上下文的寄存器，且这个上下文的寄存器对所有的方法都一样，当过程被调用时，没有可被替换的参数来生成新的切片准则。

此处，通过读取或写入的 *load* 和 *store* 操作的信息来扩展相关变量的 $ROUT(i)$ 来解决这个问题。假设 $S(v,r)$ 是一个 *store* 指令，该指令将变量 v 存储在寄存器 r 中，相应的 $L(v,r)$ 是一个 *load* 指令。$STORE(i)$ 返回指令 i 前最终存储在寄存器中的变量集合，$LOAD(i)$ 返回指令 i 后从寄存器加载的变量集合。

$$STORE(i) = \{(v,r)|\exists S_i(v,r) \rightarrow_{CFG}^{*} \text{i}, \nexists L_i(v,r) \rightarrow_{CFG}^{*} \text{i} S_j(v',r) \rightarrow_{CFG}^{*} \text{i}\} \tag{6.13}$$

$$LOAD(i) = \{(v,r)|\exists i \rightarrow_{CFG}^{*} \text{i} L_i(v,r), \nexists i \rightarrow_{CFG}^{*} L_j(v',r')_{CFG}^{*} L_i(v,r)\} \tag{6.14}$$

使用这些集合，即使所有过程使用同一套寄存器，也可以完成从形参到实参的替换。如果用栈来传递参数，可以通过指向分析来完成过程间的切片。如果栈参数是一个相关变量，那调用方和被调用方的内存地址是一样的。

6.3.2 恢复缺失的类型信息

准确的数据流分析需要对象的类型信息。在指针分析过程中，恢复了函数参数和返回值后，实例变量和协议方法的类型信息仍然是缺失的。如果这些信息在二进制代码中存在，则类型信息很容易恢复。然而，如果实例变量是通过外部库中的函数分配得来，确定变量的类型信息是非常有挑战的一件事情。

实例变量通常是通过二进制静态地址来访问。这使得我们可以找到所有引用某个变量和抽象地址的指令 [30]。类似的，二进制中，也不存储通过 Objective-C 或 Swift 的协议声明的函数参数的准确类型，虽然通过为每个参数生成抽象位置也无法进行准确的指针分析，使用这些对象仍可能识别方法调用。

6.3.3 参数反向追踪

程序切片归纳了所有影响某一参数的语句和变量，这些信息可以针对不同的执行路径进行分解。为证明某个参数符合安全规则，可以追踪所有能到达参数的路径，直到参数的首次定义。如果监测到违反规则的路径，可以指出受影响的信息流并且可以突出有问题的语句。接下来，描述在参数追踪中的查找前驱节点、抽取执行路径和反向追踪时避免环路的方法。

查找前驱节点： LLVM IR 的静态单赋值（SSA）语句提供了针对不使用指针来进行内存访问的指令的反向追踪方法。但是，如果从指针引用的位置读取了值，则无法仅通过检查语句来确定前面的存储指令。对于程序切片，有必要将位置添加到相关集合中，并向后遍历控制流程图以找到对该位置的修改。对于回溯，我们需要指定的不仅仅是相关位置。

每当将位置 l 添加到相关集合时，都会将引发该位置的语句 s 添加到另外的集合 $R_{Sources}(i,l)$ 中。集合中包含所有在语句 i 处添加了相关变量 l 的语句，并允许匹配与读取相匹配的修改指令。下面定义了如何为每条指令定义这些集合，以及相关变量的原点如何通过程序传播的。类似于静态切片的定义，指令 j 是指令 i 的直接后继 $(i \rightarrow_{CFG} j)$：

$$R_{Sources}(i,l) = \{i | i \in S_C, l \in REF(i)\} \cup \{i \in R_{Sources}(j,l), i \notin S_C\} \tag{6.15}$$

如果值是通过指针访问，那么可以利用此信息确定某个位置之前的修改。如果语句 r 从位置 l 读取值，则用于回溯读取的值的前驱定义为：

$$Pred'(r) = \{s | r \in R_{Sources}(s,l), l \in DEF(s)\} \tag{6.16}$$

包括所有可以修改引用的变量的指令的前驱集合可以描述为：

$$Pred(r) = \{s | r \in R_{Sources}(i,l), l \in DEF(s), l \in Loc\} \cup \{op | op \in Operators(i), op \in Instructions\} \tag{6.17}$$

抽取执行路径： 使用指令 i 的 $Pred(r)$ 集，可以得到一条语句的所有前驱。值 v 到起点的所有可能路径都可以看成图 $G = (V,E)$。顶点的集合 V 包含程序的所有指令，边 E 是通过递归添加所有可到达的前驱来定义。

$$E'(0) = \{(v,j)|j \in Pred(v)\} \tag{6.18}$$

$$E'(k) = \{(i,j)|(j,k) \in E^{(k-1)}, i \in Pred(j)\} \tag{6.19}$$

避免环路： 如果代码在循环中执行，路径中可能形成循环依赖。为避免分析中的循环依赖，如果发现分支的某条指令已经出现在路径中，将忽略这条分支。这样可以做到跳转中无循环依赖，因此可以达到值的原始定义。

6.3.4 切片优化

因为切片独立执行，所以可以并行切片以充分利用计算资源；因为切片准则是可合并的，所以可以避免重复切片；切片计算中间过程，如 R 是可复用的，所以通过保存这些中间过程来减少重复计算；对于过程间切片，可以针对被调用目标运用数据流分析方程来建立摘要信息，摘要信息描述某个参数是否被修改，从而可以避免对相关参数进行扩展。

6.3.5 检测规则

为对检测提供足够的灵活性，将切片准则和约束条件转化为检测规则。通过检测规则对检测问题进行描述，指导分析引擎对应用进行切片并检查相关的约束。检测规则目前的支持的上下文无关语法如下：

⟨*rule*⟩ ::= name ⟨*ident*⟩ criterion ⟨*criterion*⟩ conditions (PRE ⟨*rule*⟩ | OK ⟨*constraint*⟩)
| ⟨*rule*⟩
| ⟨*criterion*⟩ ::= name ⟨*ident*⟩ calls name ⟨*api_str*⟩ parameter ⟨*para*⟩
| ⟨*para*⟩ ::= X0 | X1 | X2 | X3 | X4 | X5 | X6 | X7 | X8
| ⟨*constraint*⟩ ::= int ⟨*int_constraint*⟩ | str ⟨*str_constraint*⟩ | call ⟨*call_constraint*⟩
| ⟨*int_constraint*⟩ ::= equal | greater | loreq | lorneq
| ⟨*str_constraint*⟩ ::= in | notin
| ⟨*call_constraint*⟩ ::= ϵ

检测规则最终以 json 格式进行封装。例如，代码**??**是在检测 GCDWebServer 安全使用时的一条检测规则。规则中包含了切片的准则和切片后的一些约束条件。

6.4 本章小结

程序切片技术在软件维护、程序调试、测试、代码理解以及逆向工程等方面有许多应用。Weiser 在自动程序抽象研究过程中提出了程序切片技术。本章在静态切片基本原理的基础上，描述了面向 iOS 应用的 IR 切片过程中所遇到的一些独特的问题。鉴于我们使用了按需进行的指向分析，导致切片分析和指向分析交叠进行，因此不方便给出具体的时间开销。

Chapter 7

结束语

iOS 应用 “大代码” 对检测工具提出了需求，本文在 LLVM [22, 42] 的基础上，参考 Andersen [1] 和 Weiser[35] 在指向分析和程序切片方面的工作，实现了面向 iOS 应用程序的切片工具。在此基础上，对现有的切片工具 [9] 进行了定量分析，并做了一些优化工作。具体来讲，在解码阶段，对无关指令进行了删除优化；在向 LLVM IR 转化阶段，对 ARC 代码进行优化；在指向分析和程序切片阶段，提出了按需进行的指向分析和程序切片方法。优化后的切片工具已经可以运行在大型 iOS 应用的分析工作中。

切片工具的实现是一项复杂的工作，实现过程面临多种方法的选择，且针对特定分析目标，要解决该目标特有的特性。目前，我们在持续进行该工具的改进工作，包括 (i) 对各种算法进行定量分析；(ii) 完善语法规则以适应新的检测需求；(iii) 持续优化以对应用进行完整分析等。

参考文献

[1] Lars Ole Andersen. “Program analysis and specialization for the C programming language”. PhD thesis. University of Cophenhagen, 1994.

[2] John P Banning. “An efficient way to find the side effects of procedure calls and the aliases of variables”. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1979, pp. 29–41.

[3] Marc Berndl et al. “Points-to analysis using BDDs”. In: *ACM SIGPLAN Notices*. Vol. 38. 5. ACM. 2003, pp. 103–114.

[4] Jiun-Liang Chen, Feng-Jian Wang, and Yung-Lin Chen. “Slicing object-oriented programs”. In: *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*. IEEE. 1997, pp. 395–404.

[5] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[6] Maryam Emami. “A practical interprocedural alias analysis for an optimizing/-parallelizing C compiler”. PhD thesis. McGill University Montreal, Québec, 1993.

[7] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. “Context-sensitive interprocedural points-to analysis in the presence of function pointers”. In: *ACM SIGPLAN Notices*. Vol. 29. 6. ACM. 1994, pp. 242–256.

[8] Manuel Fähndrich et al. “Partial online cycle elimination in inclusion constraint graphs”. In: *ACM SIGPLAN Notices*. Vol. 33. 5. ACM. 1998, pp. 85–96.

[9] Johannes Feichtner, David Missmann, and Raphael Spreitzer. “Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications”. In: *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM. 2018, pp. 236–247.

[10] Ben Hardekopf and Calvin Lin. “Exploiting pointer and location equivalence to optimize pointer analysis”. In: *International Static Analysis Symposium*. Springer. 2007, pp. 265–280.

[11] Ben Hardekopf and Calvin Lin. “The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code”. In: *ACM SIGPLAN Notices*. Vol. 42. 6. ACM. 2007, pp. 290–299.

[12] Nevin Heintze and Olivier Tardieu. “Ultra-fast aliasing analysis using CLA: A million lines of C code in a second”. In: *ACM SIGPLAN Notices*. Vol. 36. 5. ACM. 2001, pp. 254–263.

[13] Michael Hind and Anthony Pioli. “Evaluating the effectiveness of pointer alias analyses”. In: *Science of Computer Programming* 39.1 (2001), pp. 31–55.

[14] Carsten Kehler Holst. “Poor man’ s generalization”. In: *Working note, August* (1988).

[15] Susan Horwitz, Thomas Reps, and David Binkley. “Interprocedural slicing using dependence graphs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990), pp. 26–60.

[16] Bogdan Korel. “Computation Of Dynamic Slices For Programs With Arbitrary Control-Flow.” In: *AADEBUG*. 1995, pp. 71–86.

[17] Bogdan Korel and J anus z Laski. “Dynamic program slicing. I nformation Processing Letters, 29 (3): 155-163”. In: *O ctober* (1988).

[18] Bogdan Korel and Janusz Laski. “Dynamic slicing of computer programs”. In: *Journal of Systems and Software* 13.3 (1990), pp. 187–195.

[19] Bogdan Korel and Jurgen Rilling. “Application of dynamic slicing in program debugging”. In: *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*. 001. Linköping University Electronic Press. 1997, pp. 43–57.

[20] Bogdan Korel and Satish Yalamanchili. “Forward computation of dynamic program slices”. In: *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*. ACM. 1994, pp. 66–79.

[21] Loren Larsen and Mary Jean Harrold. “Slicing object-oriented software”. In: *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE. 1996, pp. 495–505.

[22] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[23] LLVM. *LLVM Alias Analysis Infrastructure*. https://llvm.org/docs/AliasAnalysis.html. Accessed Auguest 1, 2019.

[24] Weiser Mark. “Program slicing”. In: *IEEE Transactions on Software Engineering* 10.4 (1984), pp. 352–357.

[25] Esko Nuutila and Eljas Soisalon-Soininen. “On finding the strongly connected components in a directed graph”. In: *Information Processing Letters* 49.1 (1994), pp. 9–14.

[26] XABI OS. “Mach-O File Format Reference”. In: *Mac Developer Library* (2009).

[27] Karl J Ottenstein and Linda M Ottenstein. “The program dependence graph in a software development environment”. In: *ACM SIGSOFT Software Engineering Notes* 9.3 (1984), pp. 177–184.

[28] David J Pearce, Paul HJ Kelly, and Chris Hankin. “Efficient field-sensitive pointer analysis of C”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30.1 (2007), p. 4.

[29] Atanas Rountev and Satish Chandra. “Off-line variable substitution for scaling points-to analysis”. In: *ACM SIGPLAN Notices*. Vol. 35. 5. ACM. 2000, pp. 47–56.

[30] Marc Shapiro and Susan Horwitz. “Fast and accurate flow-insensitive points-to analysis”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 1–14.

[31] Marc Shapiro and Susan Horwitz. “The effects of the precision of pointer analysis”. In: *International Static Analysis Symposium*. Springer. 1997, pp. 16–34.

[32] Bjarne Steensgaard. “Points-to analysis in almost linear time”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1996, pp. 32–41.

[33] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

[34] Frank Tip et al. “Slicing class hierarchies in C++”. In: *Acm Sigplan Notices* 31.10 (1996), pp. 179–197.

[35] Mark Weiser. “Program slices: formal, psychological, and practical investigations of an automatic program abstraction method”. In: *PhD thesis, University of Michigan* (1979).

[36] Mark Weiser. “Program slicing”. In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 439–449.

[37] Robert P Wilson and Monica S Lam. *Efficient context-sensitive pointer analysis for C programs*. Vol. 30. 6. ACM, 1995.

[38] Jianjun Zhao. “Dynamic slicing of object-oriented programs”. In: *Technical-Report SE-98-119*. Information Processing Society of Japan, 1998, pp. 17–23.

[39] Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima. “Static slicing of concurrent object-oriented programs”. In: *Proceedings of 20th International Computer Software and Applications Conference: COMPSAC’96*. IEEE. 1996, pp. 312–320.

[40] 李必信. “程序切片技术及其在面向对象软件度量和软件测试中的应用”. PhD thesis. 南京大学, 2000.

[41] 李必信. *程序切片技术及其应用*. 科学出版社, 2006.

[42] 冷静文 过敏意. *LLVM 编译器实战教程*. 机械工业出版社, 2019.

[43] 陈聪明 et al. “基于包含的指针分析优化技术综述”. PhD thesis. 2011.