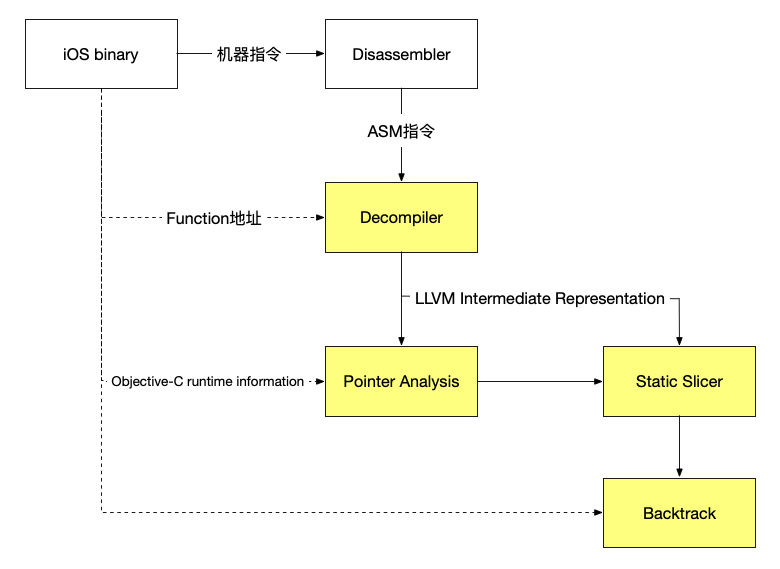
**当前工具工作流程：**

1. IPA砸壳，从中提取arm64架构的Macho二进制文件；
2. 使用llvm-dec工具将Macho进行反汇编并翻译成IR表示；
3. 使用opt工具进行IR的冗余指令结合、聚合量的标量替换、控制流简化工作；
4. 将opt优化后的IR文件及检测规则文件，传递给llvm-dfa工具进行代码逻辑安全检测。

****

整体工作流程图（黄色色块是在LLVM基础上添加的工作模块）

**llvm-dec组成**

llvm-dec主要由两个基本代码层组成：MC和DC层。

MC层提供了机器代码反汇编到MC指令的API，主要进行二进制的反汇编工作。它不是简单的线性反汇编，而是一种递归遍历反汇编，在反汇编时它从入口点开始，按照跳转和函数调用的控制流进行反汇编。反汇编的结果是一个MCModule，它由MCFunction组成，MCFunction又由MCBasicBlock组成。

DC层主要将MC指令翻译为IR表示。在其中有几个关键类：DCInstrSema、DCRegisterSema和DCTranslator。

1. DCInstrSema实现了指令级的语义。它由SemanticsEmitter TableGen后端驱动将MCInsts翻译为IR。并且负责生成的IR Module中的控制流、函数和基本块，基本上就是将MCFunction转换为IR Function。
2. DCRegisterSema生成寄存器组访问相关的代码。它生成寄存器组上下文结构类型，用它来保存和恢复寄存器状态、为各个寄存器创建局部变量、处理super-register和sub-register的语义等行为，并且翻译状态寄存器标志和条件码。
3. DCTranslator是整个翻译过程的驱动代码，从反汇编到符号化到IR的生成过程的实现。

在使用dagger时，会出现指令无法翻译的问题，如dagger文件夹下的bugs.md文件中所描述的问题，若今后遇到此类问题，可以参考相应的commit进行添加。

**llvm-dfa组成**

llvm-dfa是一个过程内数据流分析的实现。由于有函数指针和Objective-C的函数调用，因此我们首先需要分析指针的指向信息，因此我们在处理调用指令时使用了迭代的约束生成方案，并针对objc\_msgSend()方法进行了相应处理。通过指针分析我们可以得到更准确的函数调用图。

我们从规则文件中的函数API确定调用函数，以其为目标函数进行数据流分析。

llvm-slicer/lib/LLVMSlicer

├── Backtrack

│   ├── Backtrack.cpp

│   ├── Constraint.cpp //根据rule中的规则进行条件判断，后续可在此添加完善dic、array等类型的判断；

│   ├── Constraint.h

│   ├── Path.cpp

│   ├── Path.h

│   ├── PathGenerator.h

│   ├── Rule.cpp //在parseCondition方法中已经添加了字符串类型的支持，后续需要添加Dictionary和array等类型；

│   ├── Rule.h

│   └── json.hpp

├── CMakeLists.txt

├── Callgraph

│   ├── Callgraph.cpp

│   └── Callgraph.h

├── IntraDFA

│   ├── DFA.cpp

│   ├── ExternalHandler.h

│   ├── FunctionIntraDFA.cpp //在addCriterion方法中，生成待检测的初始化指令，由于原工具设计是判断某方法A的不同参数，故其PreCondition也是针对方法A；但由于我们的需求，需要找到不同的方法B，如NSPath方法（目标举例如JDEBook应用），具体逻辑见代码，此处尚未完成。为完成这个目的暂时有两种方案，一是在此处添加PreCondition，但是目前效果不好，需要在打印结果时再处理一些东西；另外是在生成Path时进行完善，但这种方式需要修改多处代码，如后续的回溯逻辑。

│   ├── FunctionIntraDFA.h

│   └── IntraDFA.cpp

├── Kleerer.cpp

├── LLVMBuild.txt

├── Languages

│   ├── LLVM.cpp

│   ├── LLVM.h

│   └── LLVMSupport.h

├── ModStats.cpp

├── Modifies

│   ├── ExternalHandler.gen

│   ├── Modifies.cpp

│   └── Modifies.h

├── PointsTo

│   ├── BuildCallGraph.cpp

│   ├── BuildCallGraph.h

│   ├── PointsTo.cpp

│   ├── PointsTo.h

│   └── RuleExpressions.h

└── Slicing

├── ExternalHandler.h

├── FunctionStaticSlicer.cpp

├── FunctionStaticSlicer.h

├── PostDominanceFrontier.cpp

├── PostDominanceFrontier.h

├── Prepare.cpp

├── Prepare.h

└── StaticSlicer.cpp

相比LLVMSlicer原目录结构，此处多添加了IntraDFA文件，即过程内数据流分析。其中IntraDFA.cpp为数据流分析主要逻辑，包括数据流及后向回溯等；FunctionIntraDFA.cpp中是IntraDFA用到的一些方法的实现，包括记录相关寄存器及语句、目标函数的查找等。

在Constraints.cpp中，可以添加规则检测逻辑，目前已实现字符串类型的比较，剩余还有dictionary、array等基本类型的描述。

Slicing文件夹中的代码因为是llvm-slicer的原实现，所以我们的llvm-dfa并没有直接用到。

llvm-slicer/lib/Analysis/Andersen

├── Andersen.cpp //在此文件中我们做了一个trick，根据rule文件中的方法名字先确定目标函数initFunctions，而后以initFunctions作为待处理对象。

├── AndersenAA.cpp

├── CMakeLists.txt

├── CallHandler

│   ├── ExternalHandler.h

│   └── ObjCRuntimeCallHandler.cpp //OC函数handler，进行OC函数解析

├── CleanUpPass.cpp

├── ConstraintCollect.cpp //建立各种约束，主要注意函数参数

├── ConstraintOptimize.cpp

├── ConstraintSolving.cpp

├── DetectParametersPass.cpp

├── ExternalLibrary.cpp

├── LLVMBuild.txt

├── NodeFactory.cpp

├── NonVolatileRegistersPass.cpp

├── ObjCCallHandler.cpp //添加OC handler

├── ObjectiveCBinary.cpp //此文件中添加了Swift的支持，解决了一些Macho文件解析问题

├── ObjectiveCClassInfo.cpp

├── SimpleCallGraph.cpp

└── StackAccessPass.cpp

在这个文件夹下面是一些基本的处理和依赖代码，包括约束优化求解、指向分析、参数检测、OC文件的解析等。

Tips

1. 输出打印可以使用errs方法；
2. 调试断点可以先添加if assert逻辑；
3. X0寄存器对应的数字是5；
4. OC property方法尚未还原，需要进行处理，还原get方法。