A 部分

任务一

reduceCFG 调用 PoRECFGReducer::inspectCFGNode 化简节点，而 reduceCFGComplete 调用 PoRECFGReducer::inspectCFGNodeComplete 化简节点。这个将会在化简循环语句中使用到。

任务二

inspectCFGNodeComplete 中的实现：

```
if(node->getSuccNum() == 1){
    //后继节点为1
    ShPtr<CFGNode> succ = node->getSucc(0);
    succs = succ->getSuccessors();
    ShPtr<Statement> gotoBody = getGotoBody(node, succ);
    reduceNode(node, gotoBody, succs, true);
```

InspectCFGNode 中的实现：

```cpp
bool PoRECFGReducer::tryReduceSequence(ShPtr<CFGNode> node) {
    /*
    * TODO: 识别并化简顺序结构，注意有些顺序结构有多个前继节点
    */
    // 获取后继节点数目
    std::size_t succNum = node->getSuccNum();
    if(!node)   return false;
    if(succNum == 1){
        ShPtr<CFGNode> succ = node->getSucc(0);
        if(succ && succ != node && succ->getPredsNum() == 1){
            //两个节点之间只有一条连线才是顺序结构
            //要排除掉自循环的节点
            ShPtr<Statement> succBody = succ->getBody();
            CFGNodeVector nextSuccessors = succ->getSuccessors();

            //不覆盖原来的语句
            reduceNode(node, succBody, nextSuccessors, true);

            return true;
        }
    }

    return false;
}
```

假设 A、B 两个节点为顺序结构，顺序结构的关键就是**保证 A 的后继节点只有一个 B，并且 B 的前继节点只有一个 A（否则会进入死循环）**。而且要排除自己指向自己的情况。

然后根据 reduceNode 的接口来获取 succBody,nextSuccessors 即可。

两种规则对比:

1. inspectCFGNode 需要考虑的情况更多，否则会造成死循环
2. inspectCFGNodeComplete 只需要考虑后续节点为一这一个条件

完成任务前后不同 sequen.bin.c 对比：

前：

```
int main(int argc, char ** argv) {
    char * v1[1];
    char * v2[1];
    _puts("Main Start");
    int64_t v3 = (float64_t)g1;
    v1[0] = (char *)&v3;
    v2[0] = (char *)*(int64_t *)*(int64_t *)0x100004010;
    _execve("/bin/bash", v1, v2);
    _puts("Done");
}
```

后：

```
int main(int argc, char ** argv) {
    char * v1[1];
    char * v2[1];
    _puts("Main Start");
    int64_t v3 = (float64_t)g1;
    v1[0] = (char *)&v3;
    v2[0] = (char *)*(int64_t *)*(int64_t *)0x100004010;
    _execve("/bin/bash", v1, v2);
    int32_t v4 = _puts("Done");
    int64_t v5 = *(int64_t *)0x100004008;
    if (*(int64_t *)v5 == *(int64_t *)*(int64_t *)0x100004008) {
        return 0;
    }
    ___stack_chk_fail((int64_t)v4);
}
```

前后基本没差别

任务三

inspectCFGNodeComplete 的实现：

```
else if(node->getSuccNum() == 2){
    //后继节点为2
    ShPtr<Expression> cond = getNodeCondExpr(node);
    ShPtr<CFGNode> succ_0 = node->getSucc(0);
    ShPtr<CFGNode> succ_1 = node->getSucc(1);

    ShPtr<Statement> gotoBody_0 = getGotoBody(node, succ_0);
    ShPtr<Statement> gotoBody_1 = getGotoBody(node, succ_1);

    ShPtr<CFGNode> succ_true = node->getSucc(0);//true分支
    ShPtr<CFGNode> succ_false = node->getSucc(1);//false分支

    if(succ_true->getPredsNum() == 1 && succ_true->getSuccNum() == 1 && succ_true->getSucc(0) == succ_false){
        succs = {node->getSucc(0)->getSucc(0)};
    }else if(succ_false->getPredsNum() == 1 && succ_false->getSuccNum() == 1 && succ_false->getSucc(0) == succ_true){
        succs = {node->getSucc(1)->getSucc(0)};
    }else if(succ_true->getPredsNum() == 1 && succ_true->getSuccNum() == 1 && succ_false->getPredsNum() == 1 && succ_false->getSuccNum() == 1 && succ_true->getSucc(0) == succ_false
    {
        succs = {node->getSucc(0)->getSucc(0)};
    }

    ShPtr<IfStmt> ifStmt = getIfStmt(cond, gotoBody_0, gotoBody_1);
    reduceNode(node, ifStmt, succs, true);
```

主要需要考虑的是当前节点的 succs 应该如何获取。

1. 如果是简单 if 语句，实现 cond 语句会跳转到 true 语句，那么下一节点就是 node->getSucc(0)->getSucc(0)

2. 如果是简单 if 语句，实现 cond 语句会跳转到 false 语句，那么下一个节点就是 node->getSucc(1)->getSucc(0)
3. 如果是 if-else 语句，那么下一节点直接就是 node->getSucc(0)->getSucc(0)

一、 If 结构的识别与化简

首先是 if 结构的判断：

```
if(node->getSuccNum() == 2 && !isSwitch(node) )
```

注意：判断条件不能有!isLoopHeader(node)，不然会识别不出来 while-if 的结构

然后根据 getIfClauseBody, getIfStmt 的接口来确定需要获取的变量。

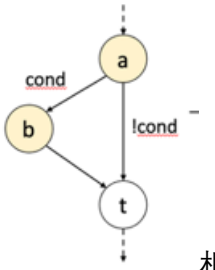
```
if(node->getSuccNum() == 2 && !isSwitch(node) ){
    ShPtr<Expression> cond = getNodeCondExpr(node);
    ShPtr<CFGNode> succ_true = node->getSucc(0);//true分支
    ShPtr<CFGNode> succ_false = node->getSucc(1);//false分支
    ShPtr<Statement> ifBody;
    ShPtr<Statement> elseBody;
    CFGNodeVector nextSuccessors;
    ShPtr<CFGNode> ifSucc;//if结构的后继节点
```

注意这里的 ShPtr<Expression> cond = getNodeCondExpr(node);必须确定是 if 结构才能使用，否则会报错。

接下来需要判断三种情况，和上述 inspectCFGNodeComplete 的实现一样：

```
if(succ_true->getPredsNum() == 1 && succ_true->getSuccNum() == 1 && succ_true->getSucc(0) == succ_false){
    //简单的if结构,分为两种情况:cond是肯定的、cond是否定的
    //简单if结构：cond是肯定的
    ifSucc = succ_true->getSucc(0);//t
    nextSuccessors = {ifSucc};
    LOG << "Try reduce if-else: " << node->getDebugStr() << "\n";

    //获得if分支代码块
    ifBody = getIfClauseBody(node, succ_true , ifSucc);
    elseBody = getIfClauseBody(node, nullptr , ifSucc);

}else if(succ_false->getPredsNum() == 1 && succ_false->getSuccNum() == 1 && succ_false->getSucc(0) == succ_true){
    //简单if结构：cond是否定的
    ifSucc = succ_false->getSucc(0);
    nextSuccessors = {ifSucc};
    cond = getLogicalNot(cond);

    //获得if分支代码块
    ifBody = getIfClauseBody(node, succ_false , ifSucc);
    elseBody = getIfClauseBody(node, nullptr , ifSucc);

}else if(succ_true->getPredsNum() == 1 && succ_true->getSuccNum() == 1 && succ_false->getPredsNum() == 1 && succ_false->getSuccNum() == 1 && succ_true->getSucc(0) == succ_false
    //if-else结构
    ifSucc = succ_true->getSucc(0);//t
    ifBody = getIfClauseBody(node, succ_true , ifSucc);
    elseBody = getIfClauseBody(node, succ_false , ifSucc);
    nextSuccessors = {ifSucc};
}
```

需要注意的是，如果是简单 if 结构，实现 cond 语句会跳转到 true 语句，那么 elseBody 的分支跳转节点是 nullptr.



根据这张图可以看出来原因是 b 和 a 都有共同的后继节点，不是 if-else 结构。

如果是简单 if 结构，实现 cond 语句会跳转到 false 语句，那么 ifBody 的分支跳转节点是 nullptr。理由同上。注意同时还要有 cond = getLogicalNot(cond);这是为了确保后面

```
if(ifBody == nullptr){     return false;}
ShPtr<IfStmt> ifStmt = getIfStmt(cond,ifBody,elseBody);
reduceNode(node, ifStmt, nextSuccessors, true);
return true;
```

ifStmt 的统一进行。

以 branch_if.bin.c 为例，

前：

```
int main(int argc, char ** argv) {
    _puts("Main Start");
}
```

```
int main(int argc, char ** argv) {
    _p   ("Main Start");
    int64_t result = 0;
    if ((int32_t)argc != 2) {
        _puts("Invalid number of arguments");
        result = 1;
    }
    return result;
}
```

后：

其他包含 if 结构的也一眼能够正确识别 if 结构。

与 Complete 的对比：

这个方法需要考虑的情况更多，但是能够更加精确地识别 if-else 结构和简单的 if 结构。Complete 需要考虑的情况更少。

二、 Switch 结构的识别与化简

遇到的主要困难是：对 API 的理解出现误差。误以为 `getSwitchStmt` 的第一个参数是 SwitchClause = std::pair<ExprVector, ShPtr<CFGNode>>的第二个参数。根据 getSwitchStmt()的第三个接口 const SwitchClauseBodyVector & clauses（switch 语句的所有分支）发现需要先循环一遍 switch 语句获得所有分支。

```
//根据SwitchClauseVector来生成SwitchClauseBodyVector
for(int i=0; i<switchClauses.size(); i++){
    //需要进行正确的初始化
    ShPtr<SwitchClauseBody> switchBody = std::make_shared<SwitchClauseBody>()
    switchBody->first = switchClauses[i]->first;

    //生成一条新的 break 语句
    ShPtr<BreakStmt> breakStmt = getBreakStmt(switchClauses[i]->second,target
    switchClauses[i]->second->appendToBody(breakStmt);

    switchBody->second = switchClauses[i]->second->getBody();
    bodyVector.push_back(switchBody);
}
```

其中因为要加上 break 语句所以找了另一个 API appendToBody。

需要注意的是，switchBody 要先进行正确的初始化，否则后面->first 会报错，空指向。

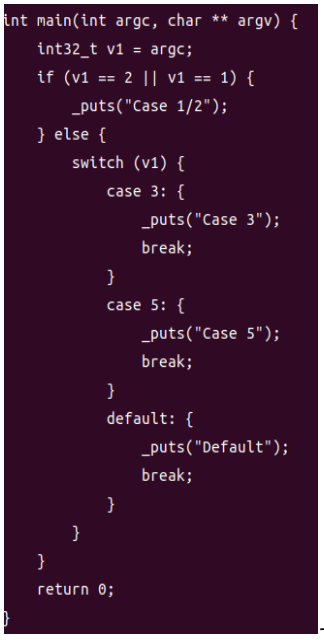
然后获得 cond 即可。

```
ShPtr<Expression> cond = getNodeCondExpr(node);//这里是node
//生成一条新的 switch 分支语句,
ShPtr<SwitchStmt> switchStmt = getSwitchStmt(node, cond, bodyVector);

reduceNode(node, switchStmt, succs, true);
```

完成前:

```
if (v1 == 2 || v1 == 1) {
    _puts("Case 1/2");
```

完成后:



```
int main(int argc, char ** argv) {
    int32_t v1 = argc;
    if (v1 == 2 || v1 == 1) {
        _puts("Case 1/2");
    } else {
        switch (v1) {
            case 3: {
                _puts("Case 3");
                break;
            }
            case 5: {
                _puts("Case 5");
                break;
            }
            default: {
                _puts("Default");
                break;
            }
        }
    }
    return 0;
}
```

与源代码语义相同。

任务四：

看到提示的 getWhileLoopStmt()需要用到 body，然后 getWhileBody()要求是自循环节点，所以想到要先考虑自循环节点。

```
//考虑自循环节点
if(node->getSuccNum() == 1 && node->getSucc(0) == node){
    ShPtr<ConstBool> cond = getBoolConst(true);
    ShPtr<Statement> body = getWhileBody(node);
    ShPtr<WhileLoopStmt> whileStmt = getWhileLoopStmt(node, cond, body);
    CFGNodeVector succs = {};

    reduceNode(node, whileStmt, succs, false);

    return true;
```

在这里遇到的困难：一开始没有正确定位自循环节点的条件 node->getSuccNum() == 1 导致一直报错。后来发现如果 node->getSuccNum() > 1，那么会被后面的 reduceCFGComplete(node)识别。

这样定位能够识别所有 while(true)语句。但是无法识别 while (i < argc)语句，只能通过 goto 来解决。

自循环节点不会跳转到其他节点，所以判断条件恒为 true，并且也没有后续节点 succs。

完成后

```
int main(int argc, char ** argv) {
    _puts("Main Start");
    while (true) {
        _puts("Hi");
    }
}
```

与源代码

```
int main(int argc, char **argv) {
    puts("Main Start");
    while (1) {
        puts("Hi");
    }
    puts("Done");
    return 0;
}
```

语义相同。

再考虑非循环节点的情况。参考给出的提示。

这里直接使用提示的 API： `reduceCFG`, `reduceCFGComplete`

需要注意的是在使用之前都要确保这个循环没有被化简过，否则会报错。

```
else if(node && isLoopHeader(node) && !isLoopReduced(node) && !isLoopHeaderUnderAnalysis(node) && !isSwitch(node)){
    enterLoop(node);

    //一定要有isLoopReduced()
    //先化简结构化语句
    while(!isLoopReduced(node)){
        if(!reduceCFG(node))
            break;
    }

    //如果还有没有被化简的节点，采用goto语句
    while(!isLoopReduced(node)){
        reduceCFGComplete(node);
    }

    leaveLoop();
    return true;
```

reduceCFG 调用 PoRECFGReducer::inspectCFGNode 化简节点，而 reduceCFGComplete 调用 PoRECFGReducer::inspectCFGNodeComplete 化简节点。把 while 语句转化成多个 if-goto 语句的结合。

生成的结果:

```
int main(int argc, char ** argv) {
    _puts("Main Start");
    int32_t v1 = argc;
    int32_t v2 = -v1;
    if (v2 < 0 == (v2 & v1) < 0) {
        return 0;
    }
    int32_t v3 = 0;
  lab_0x100003f60:
    _puts((char *)*(int64_t *)(8 * (int64_t)v3 + (int64_t)argv));
    v3++;
    int32_t v4 = v3 - v1;
    if (v4 < 0 == ((v4 ^ v3) & (v3 ^ v1)) < 0) {
        return 0;
    }
    goto lab_0x100003f60;
}
```

与源代码语义相似

```
int main(int argc, char **argv) {
    puts("Main Start");
    int i = 0;
    while (i < argc) {
        puts(argv[i]);
        i++;
    }
    return 0;
}
```

B 部分

以这个片段为例:

```
============================函数定义============================

define i64* @function_100003d7c(i32 %size) {
dec_label_pc_100003d7c:
  %0 = sext i32 %size to i64
  store i64 %0, i64* @x0

; 0x100003d7c
  store volatile i64 4294983036, i64* @_asm_program_counter
  store i64 4294983680, i64* @x16

; 0x100003d80
  store volatile i64 4294983040, i64* @_asm_program_counter
  %1 = load i64, i64* @x16
  %2 = add i64 %1, 16
  %3 = inttoptr i64 %2 to i64*
  %4 = load i64, i64* %3
  store i64 %4, i64* @x16

; 0x100003d84
  store volatile i64 4294983044, i64* @_asm_program_counter
  %5 = load i64, i64* @x0
  %6 = trunc i64 %5 to i32
  %7 = call i64* @_malloc(i32 %6)
  %8 = ptrtoint i64* %7 to i64
  store i64 %8, i64* @x0
  %9 = ptrtoint i64* %7 to i64
  store i64 %9, i64* @x0
  %10 = load i64, i64* @x0
  %11 = inttoptr i64 %10 to i64*
  ret i64* %11
}
============================定值使用============================
LLVM 值 Def:   %0 = sext i32 %size to i64
  被使用:   store i64 %0, i64* @x0
LLVM 值 Def:   store i64 %0, i64* @x0
  使用了:   %0 = sext i32 %size to i64
变量 Def: 0x5b1faebe4088
  被使用:   %5 = load i64, i64* @x0
LLVM 值 Def:
```

```
; 0x100003d7c
  store volatile i64 4294983036, i64* @_asm_program_counter
变量 Def: 0x5b1faf125b28
LLVM 值 Def:   store i64 4294983680, i64* @x16
变量 Def: 0x5b1faebe5288
  被使用:   %1 = load i64, i64* @x16
LLVM 值 Def:
; 0x100003d80
  store volatile i64 4294983040, i64* @_asm_program_counter
变量 Def: 0x5b1faf125b28
LLVM 值 Def:   %1 = load i64, i64* @x16
  被使用:   %2 = add i64 %1, 16
  使用了:   store i64 4294983680, i64* @x16
LLVM 值 Def:   %2 = add i64 %1, 16
  被使用:   %3 = inttoptr i64 %2 to i64*
  使用了:   %1 = load i64, i64* @x16
LLVM 值 Def:   %3 = inttoptr i64 %2 to i64*
  被使用:   %4 = load i64, i64* %3
  使用了:   %2 = add i64 %1, 16
LLVM 值 Def:   %4 = load i64, i64* %3
  被使用:   store i64 %4, i64* @x16
  使用了:   %3 = inttoptr i64 %2 to i64*
LLVM 值 Def:   store i64 %4, i64* @x16
  使用了:   %4 = load i64, i64* %3
变量 Def: 0x5b1faebe5288
LLVM 值 Def:
; 0x100003d84
  store volatile i64 4294983044, i64* @_asm_program_counter
变量 Def: 0x5b1faf125b28
LLVM 值 Def:   %5 = load i64, i64* @x0
  被使用:   %6 = trunc i64 %5 to i32
  使用了:   store i64 %0, i64* @x0
LLVM 值 Def:   %6 = trunc i64 %5 to i32
  被使用:   %7 = call i64* @_malloc(i32 %6)
  使用了:   %5 = load i64, i64* @x0
LLVM 值 Def:   %7 = call i64* @_malloc(i32 %6)
  被使用:   %8 = ptrtoint i64* %7 to i64
  被使用:   %9 = ptrtoint i64* %7 to i64
  使用了:   %6 = trunc i64 %5 to i32
LLVM 值 Def:   %8 = ptrtoint i64* %7 to i64
  被使用:   store i64 %8, i64* @x0
  使用了:   %7 = call i64* @_malloc(i32 %6)
LLVM 值 Def:   store i64 %8, i64* @x0
  使用了:   %8 = ptrtoint i64* %7 to i64
```

```
变量 Def: 0x5b1faebe4088
LLVM 值 Def:   %9 = ptrtoint i64* %7 to i64
  被使用:   store i64 %9, i64* @x0
  使用了:   %7 = call i64* @_malloc(i32 %6)
LLVM 值 Def:   store i64 %9, i64* @x0
  使用了:   %9 = ptrtoint i64* %7 to i64
变量 Def: 0x5b1faebe4088
  被使用:   %10 = load i64, i64* @x0
LLVM 值 Def:   %10 = load i64, i64* @x0
  被使用:   %11 = inttoptr i64 %10 to i64*
  使用了:   store i64 %9, i64* @x0
LLVM 值 Def:   %11 = inttoptr i64 %10 to i64*
  被使用:   ret i64* %11
  使用了:   %10 = load i64, i64* @x0
LLVM 值 Def:   ret i64* %11
  使用了:   %11 = inttoptr i64 %10 to i64*
```

其中，对于%0 的情况，它通过"sext i32 %size to i64"这条指令定义，然后被使用在"store i64 %0, i64* @x0"这个存储操作中，从而形成了 DU 链。

对于%5 等变量，通过 load 指令加载数据到寄存器，然后进行一系列计算和转换操作，最终将结果存储回内存中，这些变量的定义和使用关系形成了 DU 链和 UD 链。

理解：

UD 链：使用的变量在哪里被定义

UD 链：定义的变量在哪里被使用

UD 链：进行循环不变计算的检测，判定变量 x 是否未经定值就被引用，复制传播

DU 链：无用代码的删除，循环中的代码外提

DU 链和 UD 链的形成使得我们能够准确地定位变量的定义和使用位置，简化了

数据流分析的复杂性。通过这些链，能够轻松追踪数据流，识别变量间的依赖关系，从而帮助化简数据流分析。

任务二

2.2.1

1. 插件具体是如何识别形式参数和实际参数的

(1)`collectAllCalls` 调用 `createDataFlowEntry`，`createDataFlowEntry` 调用 collectDefArgs，主要看 collectDefArgs 函数。

```
if (auto* l = dyn_cast<LoadInst>(&*it))//load指令进入if
{
    auto* ptr = l->getPointerOperand();
    if (!_abi->isGeneralPurposeRegister(ptr) && !_abi->isStackVariable(ptr))
    {//不在通用寄存器和堆栈里面
        continue;
    }

    auto* use = _rda->getUse(l);
    if (use == nullptr)
    {
        continue;
    }

    if ((use->defs.empty() || use->isUndef())//UD链为空
            && added.find(ptr) == added.end())
    {
        dataflow->addArg(ptr);//添加为形式参数
        added.insert(ptr);
    }
}
```

可以看到识别形式参数的主要方式是利用定值-使用分析，查找不在通用寄存器和堆栈变量中的 load 指令，分析函数中的未初始化引用。即：指令使用的操作数，UD 链为空。

(2) `collectAllCalls` 调用 `addDataFromCall`，`addDataFromCall` 调用 collectCallArgs 函数，通过 collectStoresBeforeInstruction 收集和识别调用指令前的 store 指令，并

```
stores.erase(
    std::remove_if(
        stores.begin(),
        stores.end(),
        [values](StoreInst* s)
        {
            return values.find(
                s->getPointerOperand()) == values.end();
        }),
    stores.end());
```

且能够过滤掉与上下文不相关的存储指令。

2. 在生成优化后代码时(applyToIr)，分别如何使用识别的形式参数和实际参数

（1）修改函数定义：使用 IrModifier::modifyFunction()函数修改函数的定义。该函数会生成一个新的函数定义，并替换原有的函数定义。

（2）更新函数调用：通过调用 de.setCalledValue(newFnc)更新函数调用指令中的被调用函数值，使其指向新生成的函数定义。

（3）构建参数列表和类型：根据收集到的形式参数信息（de.args()和 de.argTypes()），构建函数的参数列表和类型列表。如果某个参数的类型未收集到，则使用 ABI 的默认类型作为该参数的类型。

3.插件除了 `collectAllCalls` 外其它部分是怎么帮助优化函数类型识别和代码生成的

collectCallSpecificTypes 函数分析调用约定。针对变参函数，尝试从函数调用中提取格式字符串（如 printf 风格的格式字符串），并使用这些信息来推断参数类型。这对于提高函数类型识别的准确性非常有帮助。

collectDefRets 和 collectCallRets 函数分别收集函数定义中的 return 指令之后的 load 指令（对于返回值类型分析）和函数调用之后的 load 指令（对于返回值的

使用分析)。这有助于确定函数的返回类型，以及如何在后续代码中使用这些返回值。

2.2.2 优化目标 1

根据提示修改 etdec/src/bin2llvmir/providers/abi/arm64.cpp。关注到 abi 部分的 if (!_abi->isGeneralPurposeRegister(ptr) && !_abi->isStackVariable(ptr))。其中

```
bool AbiArm64::isGeneralPurposeRegister(const llvm::Value* val) const
{
        uint32_t rid = getRegisterId(val);
        return ARM64_REG_X0 <= rid && rid <= ARM64_REG_X30;
}
```

检查的范围是 X0-X30，但是根据 ARM64 架构下调用约定，函数实际上只使用 X0 到 X7 之间的寄存器传递参数。所以缩小范围能够更准确地识别参数。

把 X30 改成 X7 即可得到

```
int64_t _base64_decode(int64_t a1, uint32_t a2, int64_t a3, int64_t
a4, int64_t a5, int64_t a6);
int64_t _base64_encode(int64_t a1, uint32_t a2, int64_t a3);
int64_t _test(int64_t a1, int64_t a2, int64_t a3, int64_t a4);
```

```
// General purpose registers
createRegister(ARM64_REG_X0, _regLt);
createRegister(ARM64_REG_X1, _regLt);
createRegister(ARM64_REG_X2, _regLt);
createRegister(ARM64_REG_X3, _regLt);
createRegister(ARM64_REG_X4, _regLt);
createRegister(ARM64_REG_X5, _regLt);
createRegister(ARM64_REG_X6, _regLt);
createRegister(ARM64_REG_X7, _regLt);
createRegister(ARM64_REG_X8, _regLt);
createRegister(ARM64_REG_X9, _regLt);
createRegister(ARM64_REG_X10, _regLt);
createRegister(ARM64_REG_X11, _regLt);
createRegister(ARM64_REG_X12, _regLt);
createRegister(ARM64_REG_X13, _regLt);
createRegister(ARM64_REG_X14, _regLt);
createRegister(ARM64_REG_X15, _regLt);
createRegister(ARM64_REG_X16, _regLt);
createRegister(ARM64_REG_X17, _regLt);
createRegister(ARM64_REG_X18, _regLt);
createRegister(ARM64_REG_X19, _regLt);
createRegister(ARM64_REG_X20, _regLt);
createRegister(ARM64_REG_X21, _regLt);
createRegister(ARM64_REG_X22, _regLt);
createRegister(ARM64_REG_X23, _regLt);
createRegister(ARM64_REG_X24, _regLt);
createRegister(ARM64_REG_X25, _regLt);
createRegister(ARM64_REG_X26, _regLt);
createRegister(ARM64_REG_X27, _regLt);
createRegister(ARM64_REG_X28, _regLt);
```

在 arm/arm64.cpp 中，通用寄存器只有 X0~X28，所以范围只需要保持在这之间即可，把 X30 改成 X28 也能够解决问题。

2.2.3 优化目标 2

分析.ll 文件中 byteswap32_to_host 函数的调用者发现它自身并不接受任何参数。

2.3 任务 3：改进反编译器

任务 2.2.2 中修复 bug 后，参数仍然存在问题，无法识别出来 const