第三章　混淆Pass的设计与实现

在介绍混淆pass的设计内容之前，我们首先需要了解基本块和控制流。

定义 3.1 (基本块). 在编译器构造中，一个基本块是一个直线的代码序列，它除了入口外没有分支，除了出口之外没有分支[13, 14]。

定义 3.2 (控制流). 在计算机科学中，控制流程是执行或评估命令式程序的各个语句，指令或函数调用的顺序。

3.1 不透明谓词

在计算机编程领域，**谓词**–一个计算值为“true”或者“false”的表达式。**不透明谓词**[8,15,16]则意味着程序开发者提前知道该表达式计算的结果，但是由于各种原因，该表达式仍然需要再运行时进行计算评估。

不透明谓词已经被用于水印，因为它可以在程序的可执行文件中被识别。它还可以用于防止编译器过度优化。

3.1.1 数学表示

不透明谓词的数学表示源于Collberg的论文[6]。

通俗地说，如果一个变量V有一些性质q（这些性质混淆器知道，但是去混淆器判断是很困难的），则称它是不透明的。

在一个程序中，如果变量V在点p有一个性质q(该性质在混淆时是已知的)，该变量V在点p是不透明的。我们用V^q来表示，或者如果p在上下文中是很清晰的话，我们可以使用V^q来表示。

如果一个谓词P，它的结果在混淆时是已知的，我们就称它在点p是不透明的。如果P总是计算结果为False(True)，我们记做$\mathbf{P^F}$($\mathbf{P^T}$)。当然了，如果P在p点的值有时为True，有时为False，我们就用$\mathbf{P}^?$来表示。并且，如果p在上下文中很清楚的话，我们会将它省略。

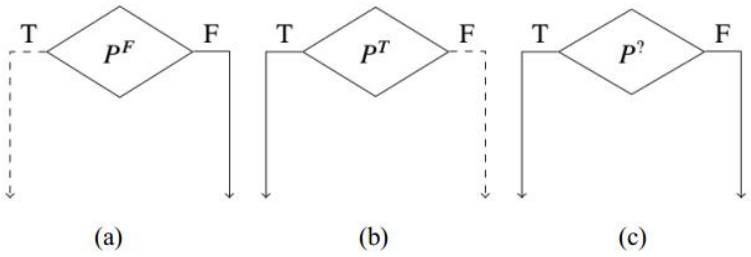


图3–1　不透明谓词的三种表示

3.1.2 数论不透明谓词

基于已经有的数论中同余的理论，可以帮助我们产生单向不透明谓词。而且它是准确可靠的。表3-1包含4个我们之后会用到的数论结果。

表3–1　同余表达式

$\forall x \in z$	$2 \mid x(x+1)$
$\forall x \in z$	$3 \mid x(x+1)(x+2)$
$\forall x \in z$	$4 \mid x^2(x+1)^2$
$\forall x \in z$	$19 \mid 4x^2+4$

因为在接下来的 Pass 实现中会使用到不透明谓词。我们这里简述下不透明谓词基本块的创建流程的函数实现方案。

1. 创建基本块，命名为obf.name。
2. 生成随机数，因为之后我们要随机在4个数论不透明谓词中随机选择基本块创建。
3. 创建分支结构，根据生成的随机数选取指定的不透明谓词，并生成IR指令。
4. 根据传入的参数构造条件跳转IR指令。
5. 返回新创建的obf.name基本块。

3.2 扩展判断条件Pass

这是实现的第一个混淆Pass。用于对*if*判断条件做处理。

3.2.1 设计

我们的方案就是在if判断条件处添加一个计算结果恒为true的表达式。设原始的判断条件为if.cond，不透明谓词的表达式的条件为P^T，我们的目的是使混淆后的条件cond为if.cond&&P^T（因为P^T恒为true，所以对原始程序的执行结果没有影响。

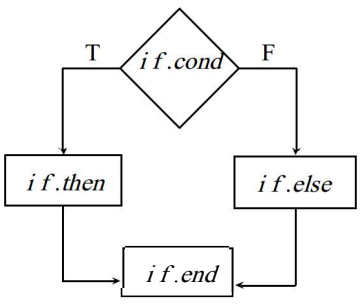


图3–2　if控制流图

扩展条件，添加P^T后的if流程图。

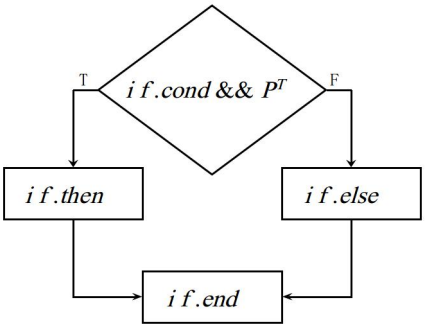


图3–3　扩展后的if控制流图

下面开始介绍关于该设计的具体LLVM Pass实现方案。

3.2.2 实现

我们的重点就是分析LLVM IR是如何处理&&判断条件的。然后添加我们的P^T控制流到原始判断流程中。

以一个取两个数值中更大的数的c语言函数为例：

```
int max(int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

生成的 LLVM IR 的控制流如图 3-4。

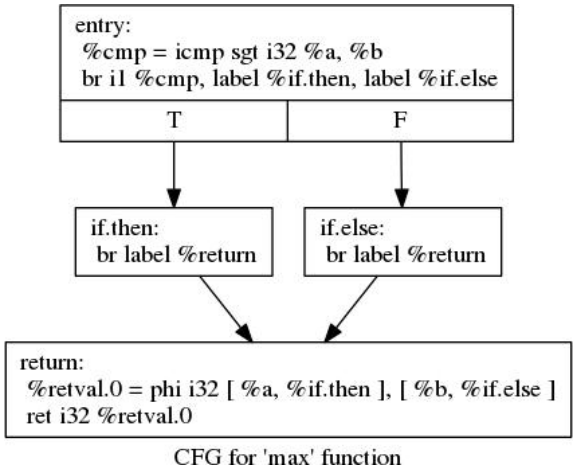


图3–4　max函数生成的LLVM IR控制流图

在 LLVM IR中，实现条件“与”操作，并不是通过特定的指令，而是通过添加判断控制流实现的。例如，当我们在上面的控制流中添加一个数论不透明谓词判断表达式时，生成的LLVM IR控制流如图3-5。

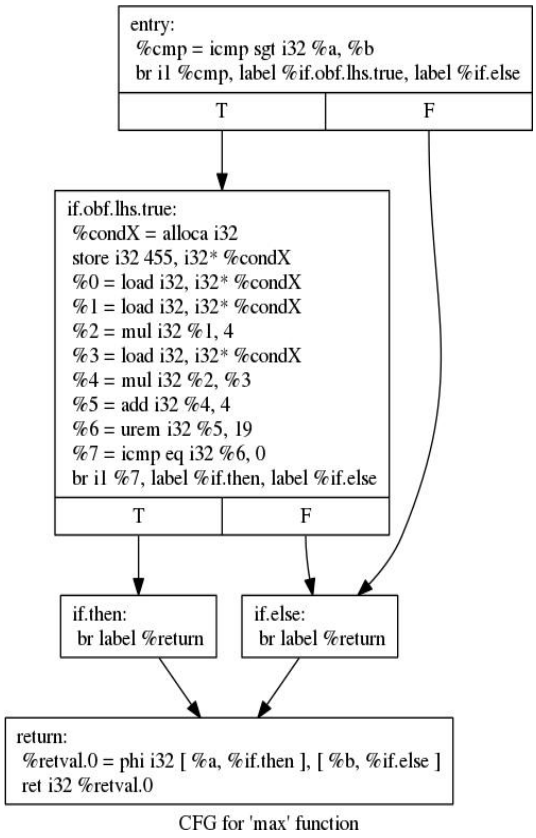


图3–5　max函数扩展判断条件生成的LLVM IR控制流图

实现方案：依次遍历每个函数的所有基本块BB，以基本块为单位做以下处理。

1. 判断当前BB的最后一条指令是否为条件跳转指令，如果是，执行下一步；不是，直接执行下一次循环。
2. 创建我们的不透明谓词P^T基本块ObfBB，ObfBB为True时跳转到BB为True时的基本块；ObfBB为False时跳转到函数最后一个基本块。
3. 将BB的最后一条条件跳转指令为True时要跳转的基本块改变为跳转到ObfBB。

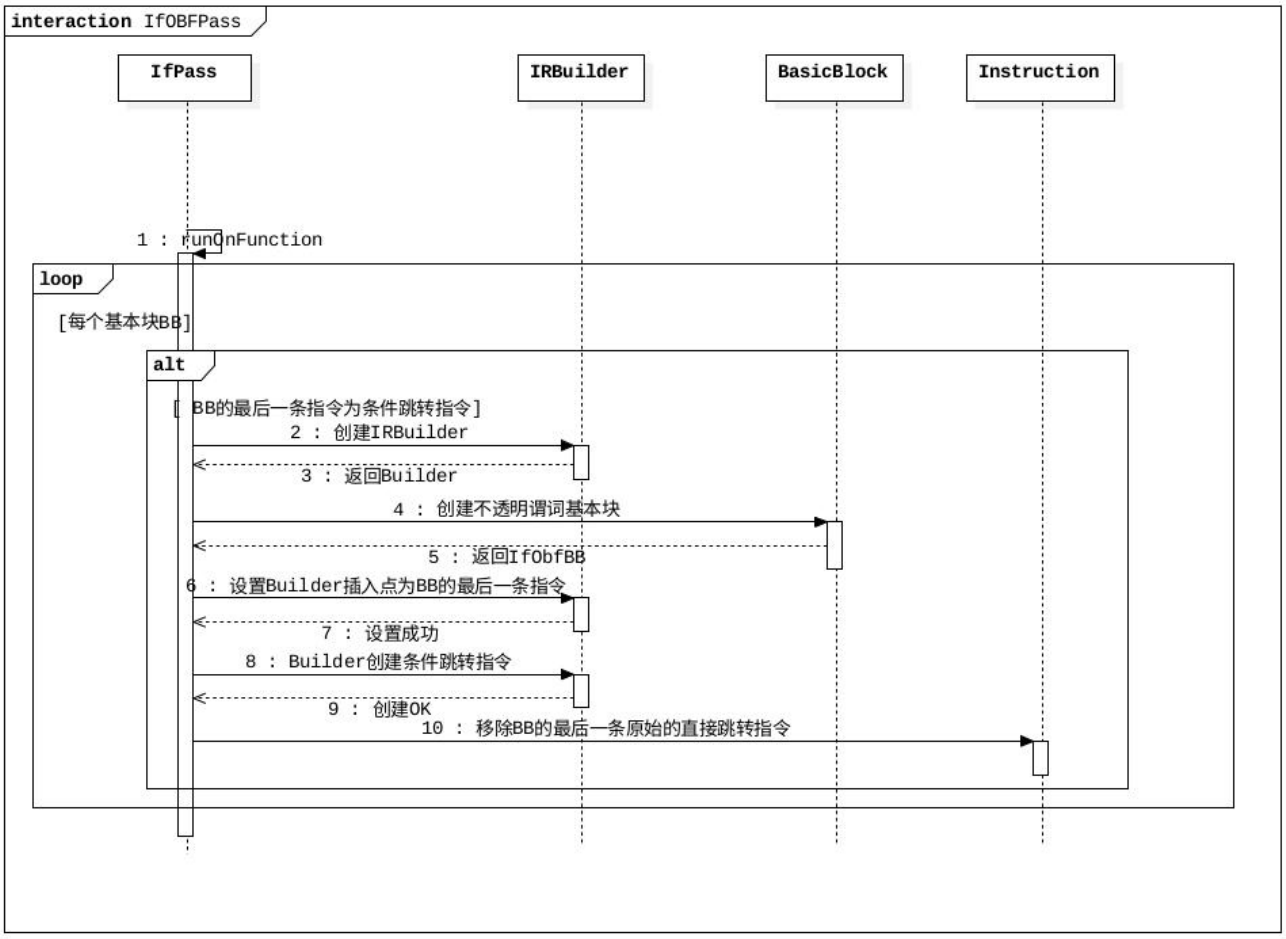


图3–6　ifPass实现的时序图

3.3 插入冗余控制流Pass

在LLVM中，所有的控制流都是由基本块组成的，所以我们在插入冗余控制流时也是基于基本块来做操作。

接下来，我们会依次讨论针对顺序基本块和循环控制流,我们要如何设计方案来插入冗余控制流。当然，本质上上一节的内容也是属于插入冗余控制流，不过体现在高级语言上，它更类似于扩展判断条件，所以独立列为一节。

3.3.1 设计

1. 针对单个基本块

假设我们当前要处理基本块S_0，那么我们首先将其拆分为S_A和S_B两个基本块。显而易见，S_A和S_B是顺序相连的，如图3-7。

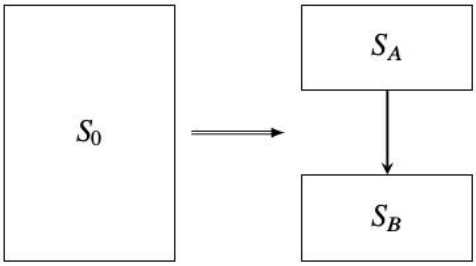


图3－7　　S_A连接S_B基本块

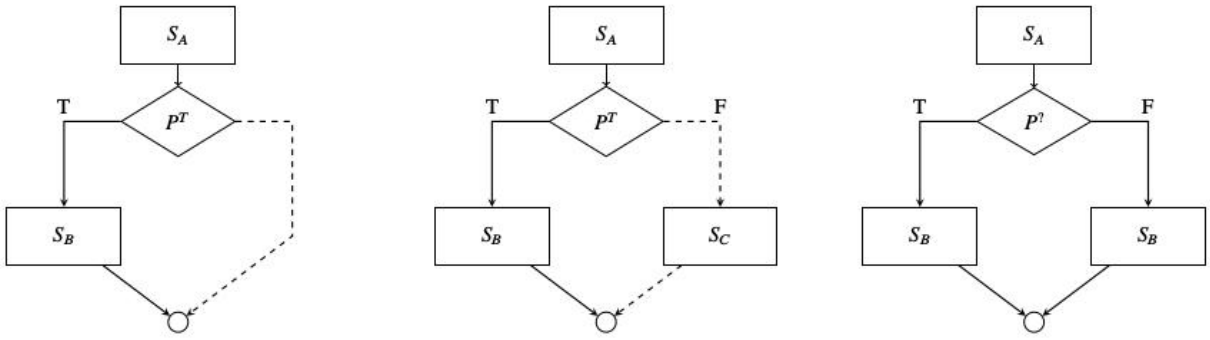
我们要插入冗余控制流使原始流程（即依次执行S_A和S_B）的结果不改变。我们采用了插入不透明谓词的方式。

如何插入这里设计了三种方式：

(a) 在S_A和S_B之间插入一个恒为True的条件表达式，并设置条件判断为True时，接着执行S_B中的指令。

(b) 我们依旧插入了一个恒为True的条件表达式，且S_B的处理和上一点一样，唯一不同的是我们引入了S_C基本块（这是我们构造的冗余基本块）它不会执行。

(c) 这里我们引入了一个运行时确定条件结果的条件表达式$P^?$，这会极大的提高静态逆向分析的难度。但是为了不改变原始程序执行流程，我们要在条件判断为True和False时都执行S_B基本块。



(a)　　(b)　　(c)

图3–8　向S_A和S_B基本块中插入冗余控制流

注意到我们这里并没有使用到P^F，因为我们完全可以对一个条件表达式取反得到它取反后布尔值，所以就不特意取P^F的不透明谓词了。

2. 针对for循环

以一个求累积和的算法为例：

```
int forTest(int a){
    int result = 0;
    for (int i = 0; i <= a; i++) result
        += i;
    return result;
}
```

可以看到，一个for循环可以由四个部分组成

- 初始化语句：inti=0;
- 循环判断条件：i<=a;
- 循环体：result+=i;
- 循环迭代表达式：i++;

事实上，在由Clang对for循环结构生成的LLVMIR也是包含这四个部分。

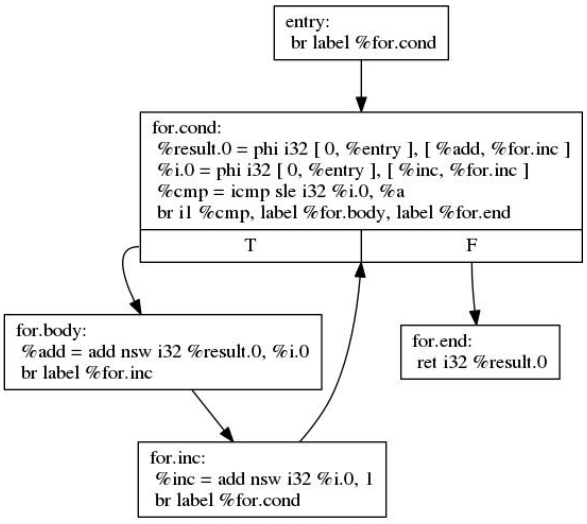


图3–9　一个for循环结构生成的LLVMIR控制流图的例子

我们将图3-9抽象成五个基本块组成的for循环控制流：

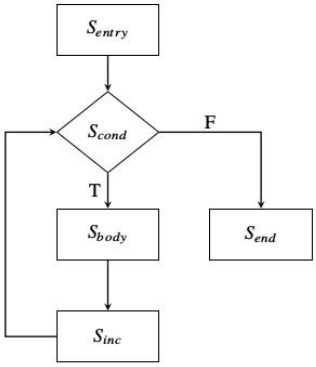


图3－10　　for循环IR的控制流图

为了方便我们分析，我们将该图化为不可归约的。即将S_{entry}和S_{cond}合并为一个节点，S_{body}和S_{inc}合并为一个节点，如图3-11。

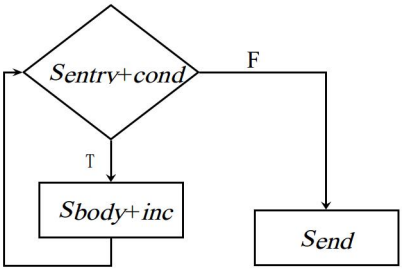


图3－11　　将图3-7化为不可归约图

现在我们可以插入我们的冗余控制流的。仍然采取插入不透明谓词的方式，由于P^T的计算结果恒为True，所以插入P^T后不影响原始程序的执行流程。并且我们还在P^T的False分支插入了一个冗余基本块S_{obf}（该基本块不被执行，只是为了增加程序的静态分析难度），如图3-12。

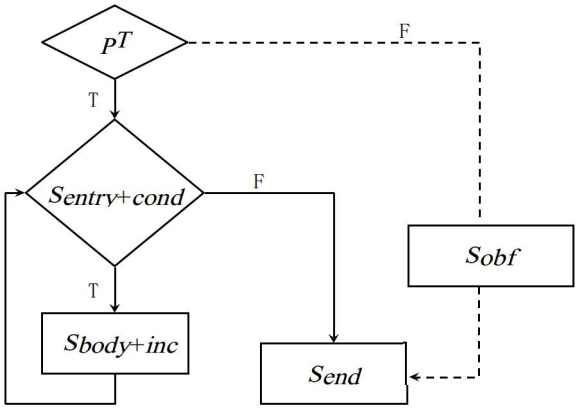


图3–12　向循环结构中插入冗余控制流1

显然地，上面这种方法可以提供静态分析的难度。但是如果逆向分析人员将P^T从原始程序中移除，那么我们的工作就不起作用了。为了应付该攻击，我们又设计了一种新的方案图3-13。

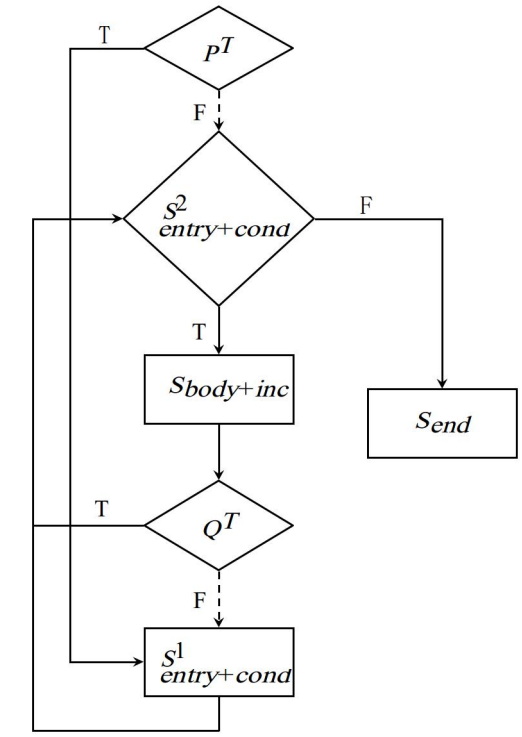


图3 - 13　　向循环结构中插入冗余控制流2

3.针对while循环

while循环与for循环类似。仍然以一个while循环为例:

```
int whileTest(int a){

    int result;

    while(a > 1) {
        int b = a*a;

        result += a;
        a--;
    }
    return result;
}
```

生成的LLVM IR:

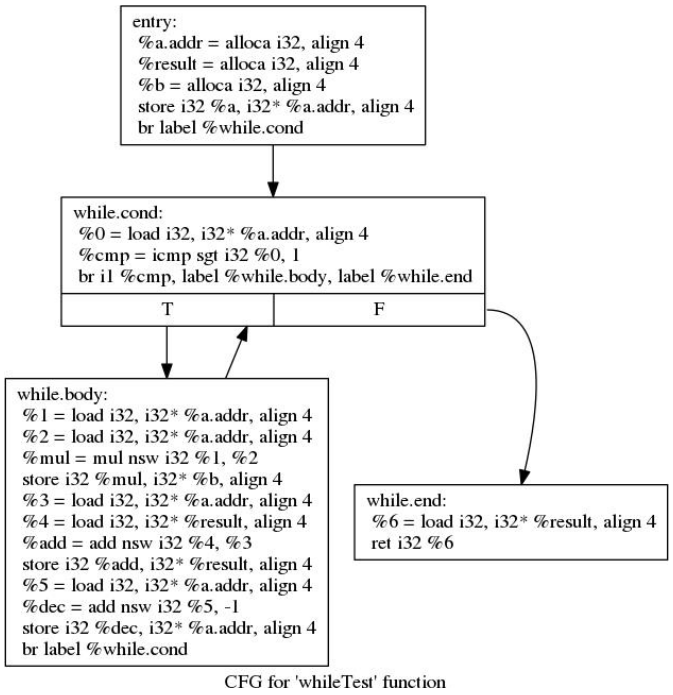


图3－14　　一个while循环结构生成的LLVMIR控制流图的例子

将LLVM IR控制流图抽象为四个基本块S_{entry}，S_{cond}，S_{body}，S_{end}。

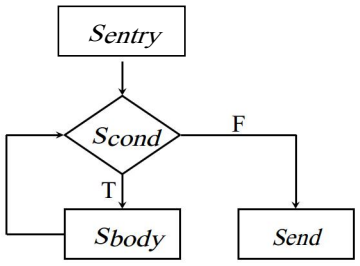


图3–15　while循环IR的控制流图

可以看到，图3-15与图3-11类似。所以我们可以对while循环采用与for循环相同的方案来进行混淆。由于设计方案类似，就不具体阐述了，参考for循环的设计方案即可。

3.3.2 实现

1. 针对单个基本块

根据对单个基本块混淆方案的设计可以了解到：

a. 我们首先会将基本块进行分裂，所以基本块指令所包含的指令数目必须大于2条。

b. 我们会构造不透明谓词插入到原始控制流中（关于不透明谓词的构造请参考3.1节的内容）。并且为了该方案不与其他方案冲突，且保证该方案实现的正确性。

我们只对两种基本块做处理：

- 基本块的最后一条指令为直接跳转指令。
- 基本块的最后一条指令为返回指令。

实现方案：依次遍历每个函数的所有基本块BB，以基本块为单位做以下处理。

1) 判断该基本块的最后一条指令是否符合要求（为直接跳转指令或返回指令）。是，则进行下一步操作；否，则直接循环continue处理下一个BB。
2) 根据所包含指令的数目从中间拆分BB，拆分后的基本块命名为“single.split.BB”。
3) 创建不透明谓词基本块，命名为"single.obf"。
4) 更改BB的最后一条跳转指令，跳转到"single.obf"基本块。

2.针对循环基本块

根据从for循环基本块的处理方案可以了解到核心内容为:

- 找到循环条件基本块for.BB，然后找到循环条件基本块的两个前基本块：BB（循环控制流的前基本块）和for.inc.BB（循环迭代基本块）。
- 创建两个不透明谓词基本块，根据设计方案插入到循环控制流中。

实现方案：依次遍历每个函数的所有基本块BB，以基本块为单位做以下处理。

1) 判断当前基本块BB的直接跳转基本块是否为for.cond。若是，执行下一步；不是，直接中止本轮循环，执行下一次循环。
2) 找到循环结构的for.inc基本块。
3) 分裂BB，分裂后的基本块命名为"for.cond.pre.split.BB"。
4) 创建不透明谓词基本块，命名为"for.obf.P"。
5) 更改for.cond.pre.split.BB的最后一条跳转指令为跳转到for.obf.BB基本块。
6) 创建不透明谓词基本块，命名为"for.obf.Q"。
7) 更改for.inc基本块的最后一条指令，改变为跳转到for.obf.Q基本块。

我们在具体实现该Pass时，分别将对基本块进行处理和for循环处理分为两个函数。

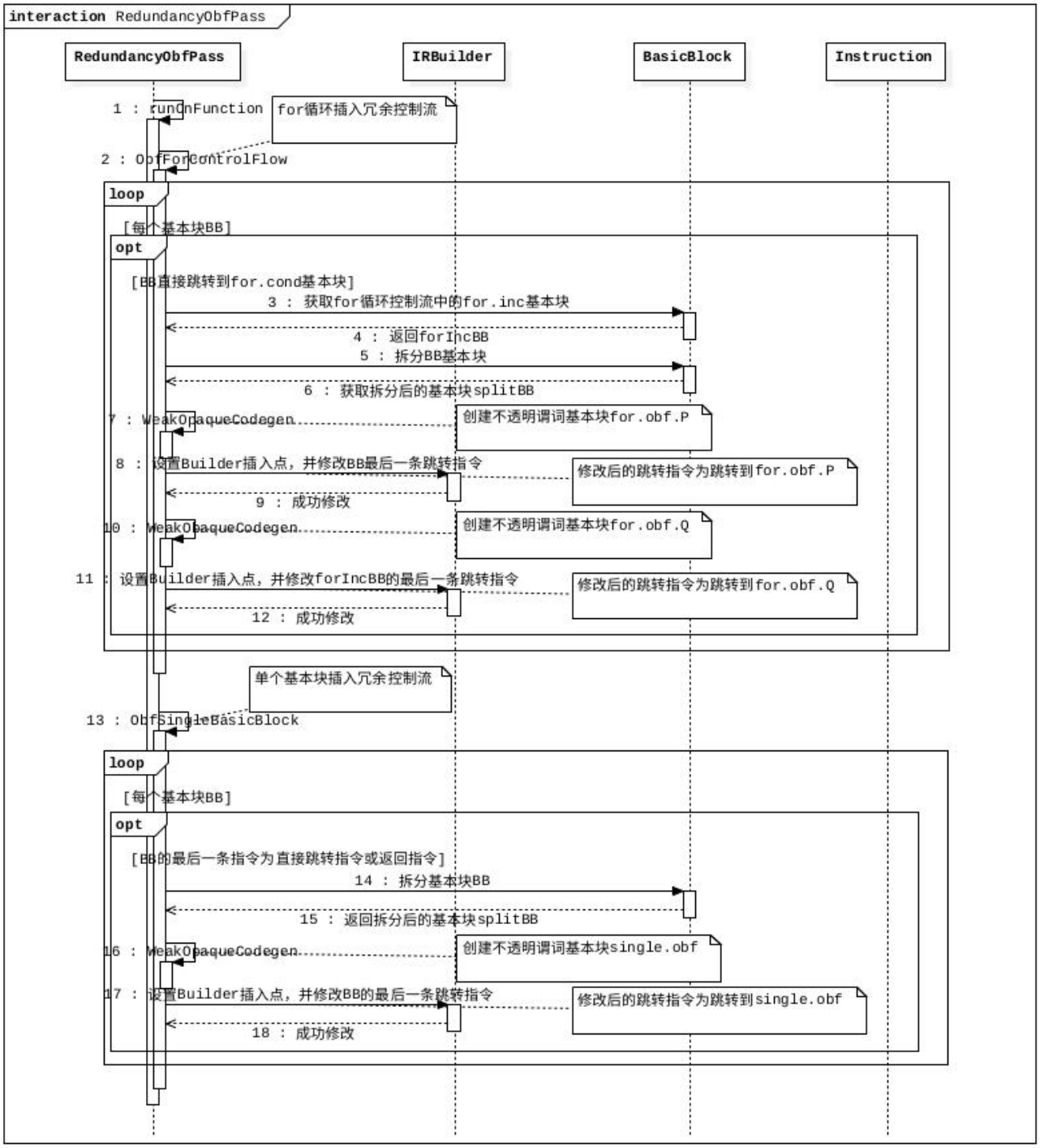


图3–16　RedundancyObfPass实现时序图

3.4 控制流平展化Pass

控制流平展化，最初是在ChenxiWang的论文[9,10]中提出。论文描述了我们可以将控制流转换为如下的通用结构：

```
while ( )
{
    switch( )
    {

    }
}
```

3.4.1 设计

设计方案：我们可以设置循环条件为1，即循环一直执行。然后通过switch的派遣变量swVar来决定要执行哪一个分支。当然了，我们会插入很多不执行的无用分支，用来提高代码的安全性。

在设计我们的方案时，有下面几个点需要我们思考：

- 由于switch结构中有很多分支，原始控制流的分支会被执行，而人工构造的冗余控制流不会执行。那么应该如何拆分原始控制流分散在多个case分支中？
- 为了保证控制流平坦化后原始程序控制流的运行顺序不改变，我们应该如何确定派遣变量swVar的值？

这两个问题是我们首先要解决的，当然了，解决方案有很多。针对这两个问题，我采取的方案是：

1. 为了保证平坦化后核心执行流程不至于太多，来很大程序上影响到程序的执行效率。所以决定根据控制流类型分为直流控制流、if判断控制流、for循环控制流、while循环控制流、do-while循环控制流和switch分支控制流。每个case分支都只执行上述6个控制流中的一个。
2. 为了使设计尽可能的简单。我们决定在swVar为奇数时，执行真实控制流；swVar的取值依次是1,3,5,7....

假设我们要处理的函数为下面的flat_test函数。

```
void flat_test()
{
    a = b + c;
    for(i = 0; i < a; i++)
    {

    }
    switch( )
    {
        case...

        case...

    }
}
```

那么它经过我们设计的平坦化Pass处理后,flat_test的执行流程变成了：

```
  void flat_test()
{
    swVar = 1;
    while(1)
    {
        switch(swVar)
        {
            case 1:
                a = b + c;
                break;

            case 2:
                人工构造冗余基本块1
                break;

            case 3:
                for(i = 0; i < a; i++)
                {

                }
                break;

            case 4:
                人工构造冗余基本块2
                break;

            case 5:
                switch( )
                {
                    case...

                    case...

                }
                break;

            case 6:
                人工构造冗余基本块
                3
                break;
            default:
                人工构造冗余基本块
                4
    }
}
```

3.4.2 实现

根据我们前面的设计方案，可以将Pass实现流程分为下面4步。

1. 将被处理函数的最外层全部控制流根据前面的设计方案分为6类（直流控制流、if判断控制流、for循环控制流、while循环控制流、do-while控制流、switch-case控制流），记录在向量BBVec中（并标记每个控制流的起始基本块和结束基本块）。
2. 使用LLVM IR构造最外层while循环控制流和主控制switch-case分支，并设置派遣变量swVar。
3. 按照顺序将向量BBVec中保存的真实控制流插入到主switch-case的奇数分支中。
4. 在主switch-case的偶数分支插入不透明谓词基本块。

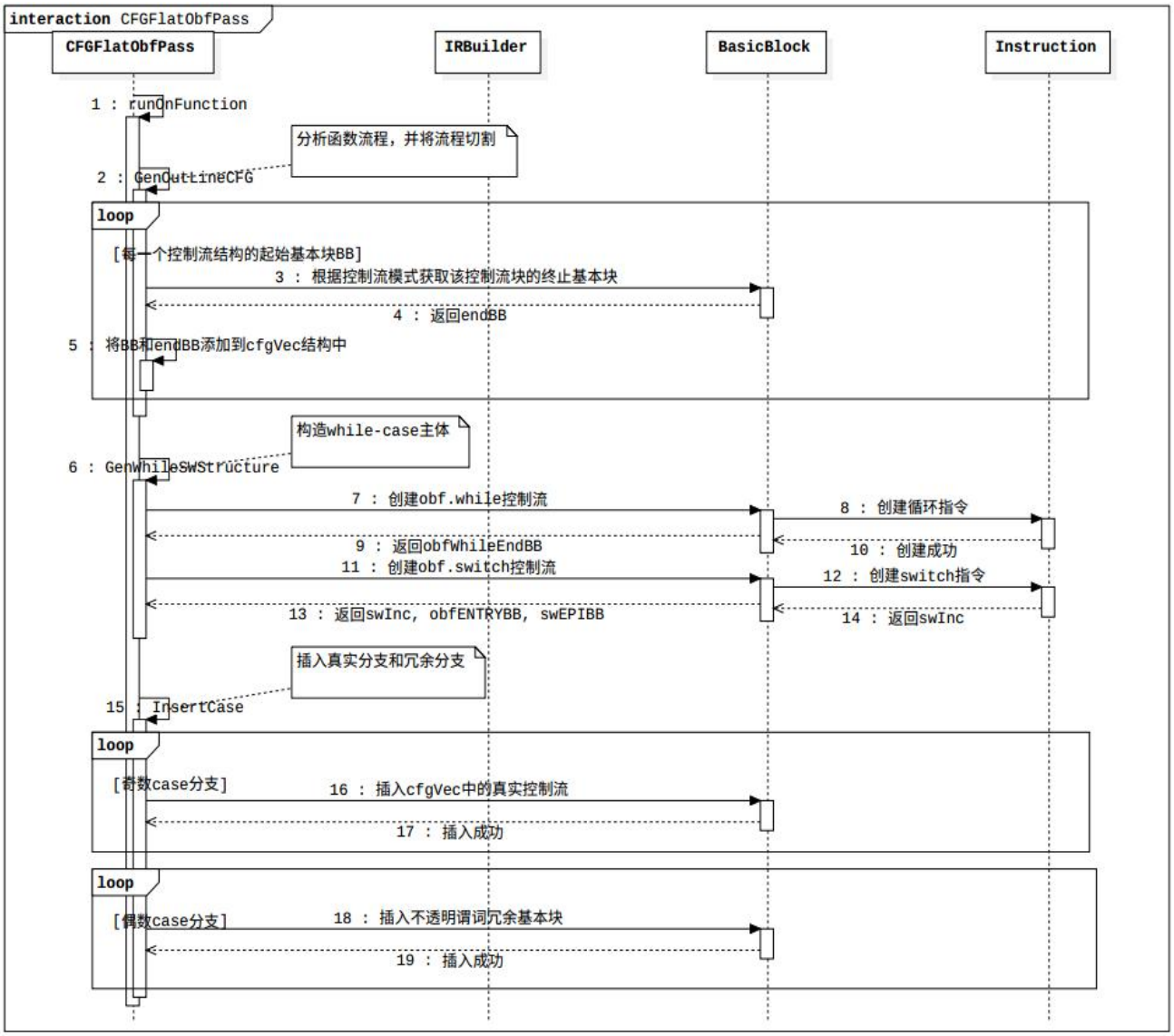


图3–17 CFGFlatObfPass实现的时序图