
利用GCC进行代码的自动化审计
CoolQ/Redpig/Fedora
X'Con2006

内容简介

- 软件发展面临的问题
- 自动化（静态）分析的整体框架
- 一次尝试BDL(Bug Descprition Language)
  - 整体设计
  - 模块分解
  - 功能及性能的改进

软件发展面临的问题

- 开发工具越来越多，门坎越来越低
  - 代码质量不好控制
- 软件的逻辑越来越复杂
  - 对设计人员对要求越来越高，程序逻辑越来越难处理
- 软件的体积越来越大
  - Bug-fix的代价在增加
- 攻击技术多种多样
  - 稍微有点错误就会被别人钻空子

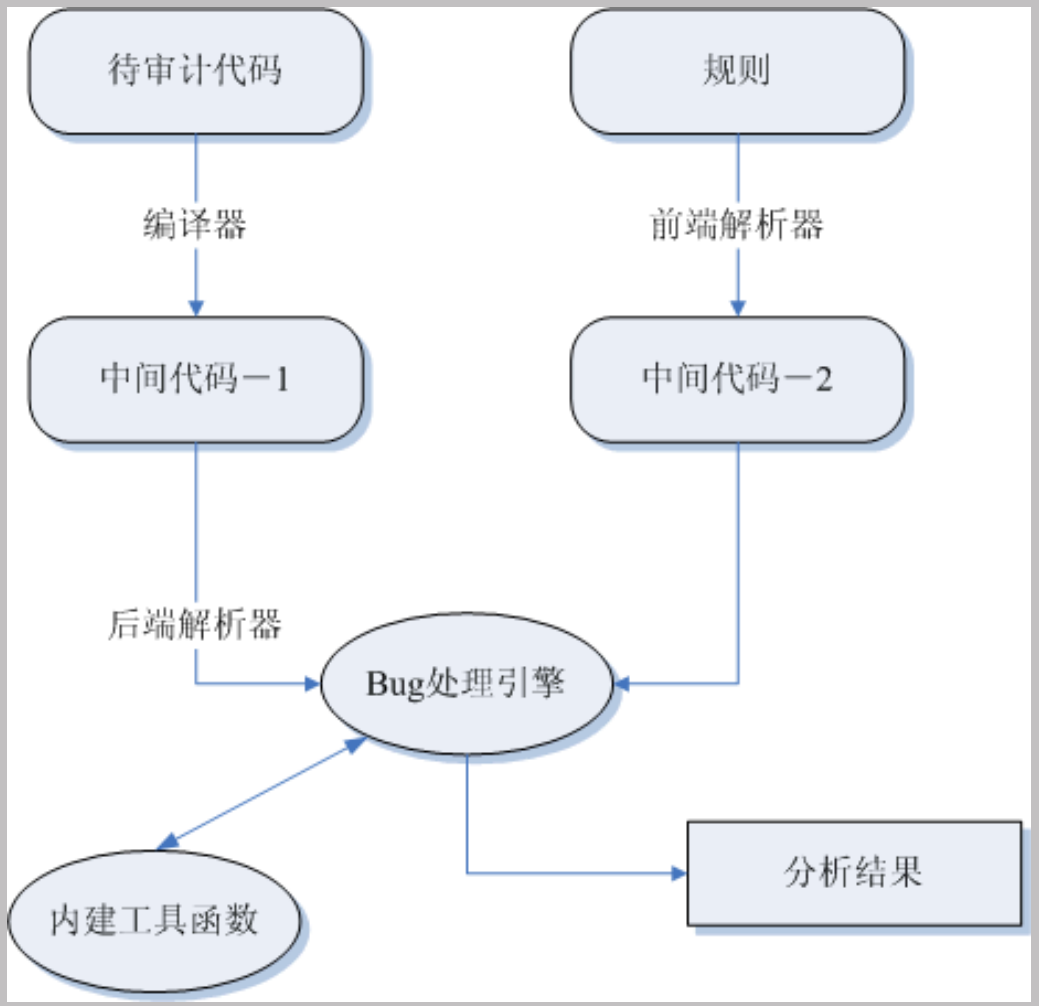
我们面临的挑战

- 能否进行Bug的自动化审计
- 速度快慢
- 误报率与漏报率高低
- 自定义Bug规则
- 克服静态分析的缺点
  - 指针
  - 过多的数据流和控制流分支

自动化分析器的整体框架

- 语言的词法分析器及语法分析器
  - 对待审计代码进行词、语法分析
- Bug规则的词法分析器及语法分析器
  - 对Bug规则进行词语、法分析
- 中间代码的存取
  - 保存词、语法分析的中间结果，并提供的访问手段
- 分析控制引擎
  - 根据规则对两种中间代码进行操作
- 工具函数
  - 提供规则匹配、Bug回溯、错误输出等功能

自动化分析器的整体框架(图示)



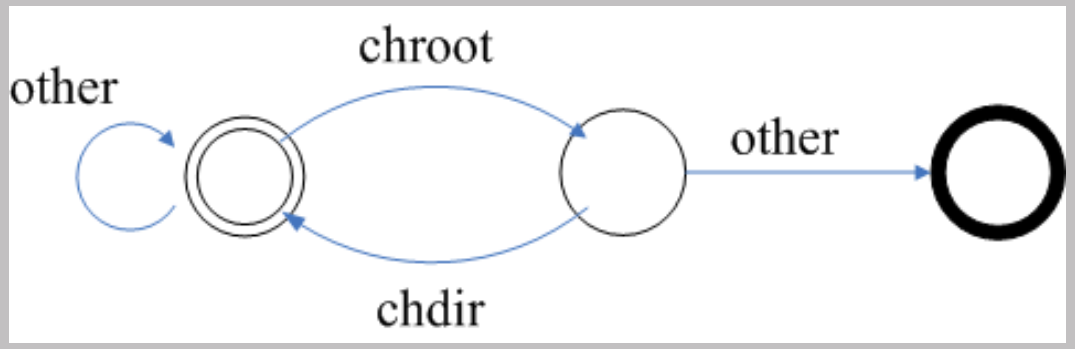
一次尝试——BDL

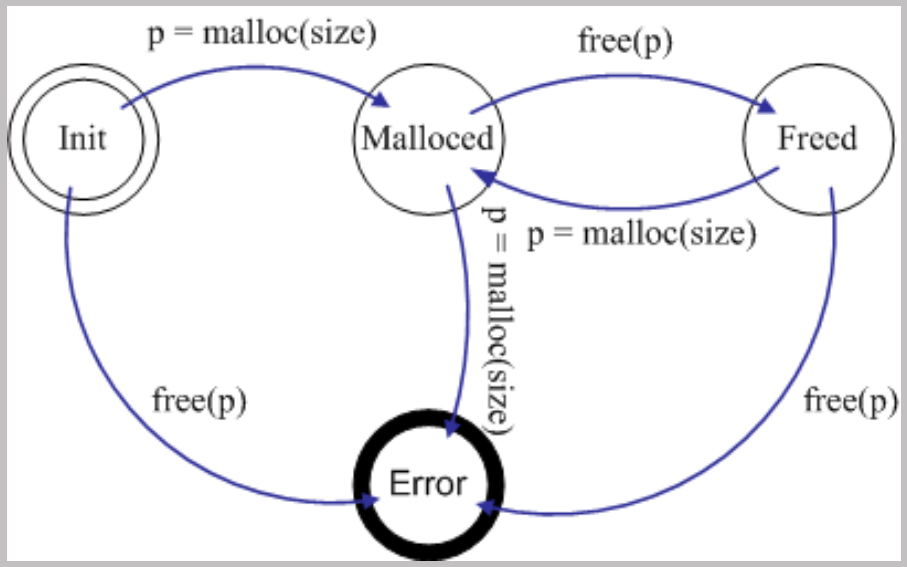
- BDL – Bug Description Language
- 基于GCC＋Flex＋Bison
  - Bug规则由Flex ＋ Bison自定义的语言描述
  - 待审计代码由GCC进行解析
- 追踪单变量的状态，使用自动机引擎进行状态转换
- 支持控制流和简单的数据流
- 多种方式的优化，提高检测速度

BDL可以描述的逻辑Bug

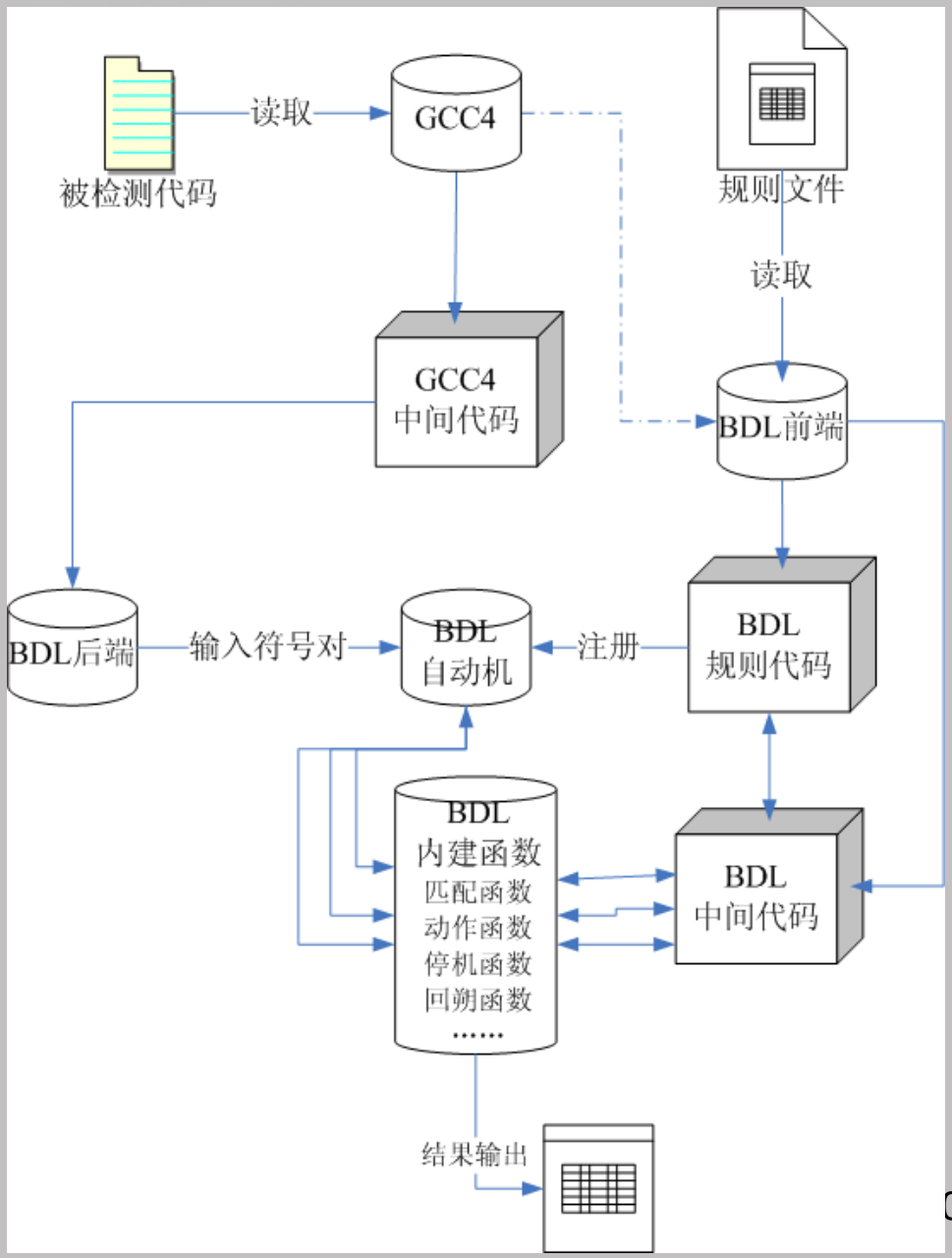
- 一定要 / 不要做X
  - 内核中不能出现浮点运算
- 做了X后需要 / 不能做Y
  - 内存对分配与释放
  - 加锁与解锁
- 在做X之前一定要 / 不能做Y
  - strcpy(dst, src)
  - printf(str)

状态转换举例

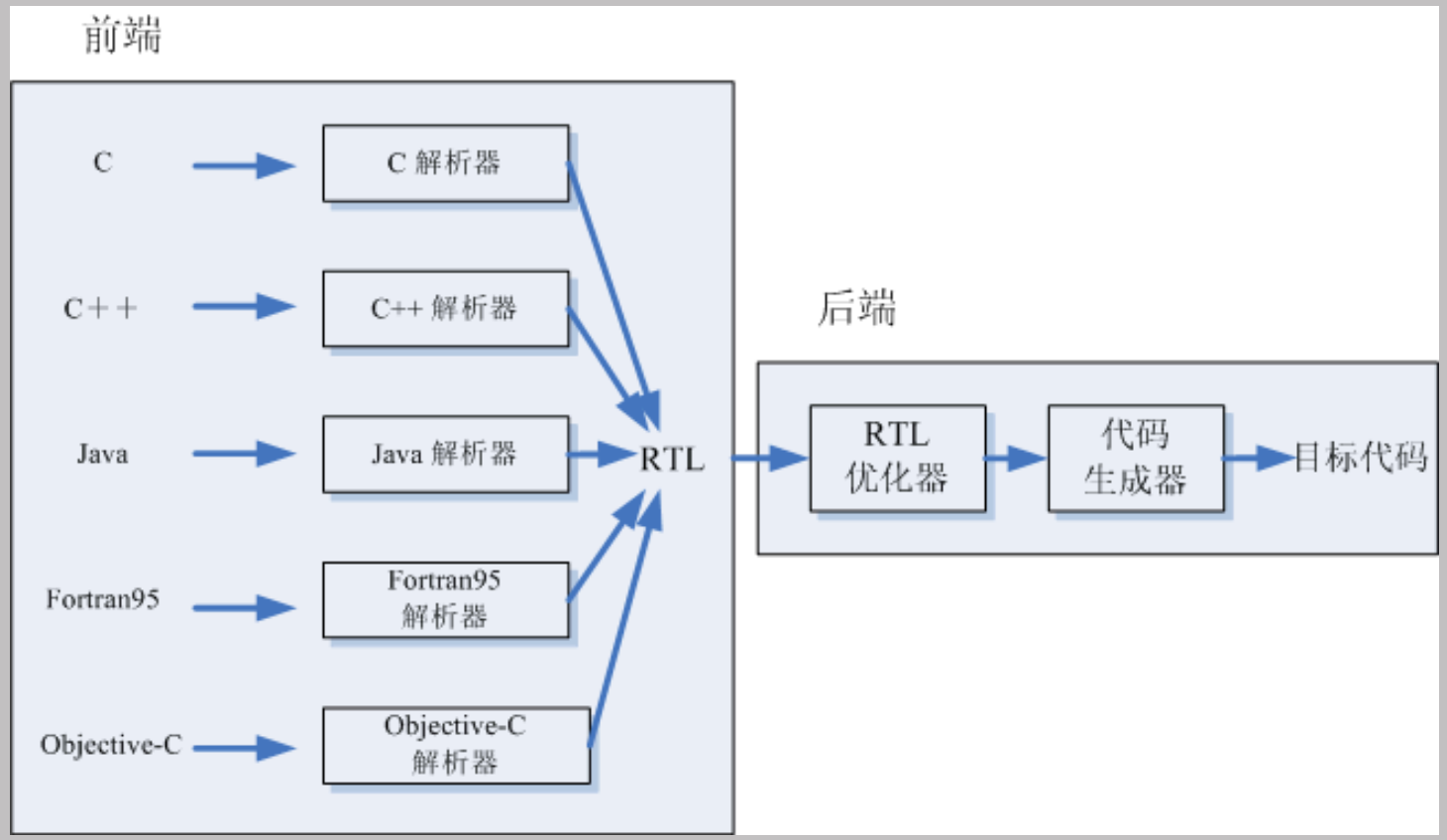

chroot
other
other
chdir


p = malloc(size)
free(p)
Init
Malloced
Freed
p = malloc(size)
p = malloc(size)
free(p)
free(p)
Error

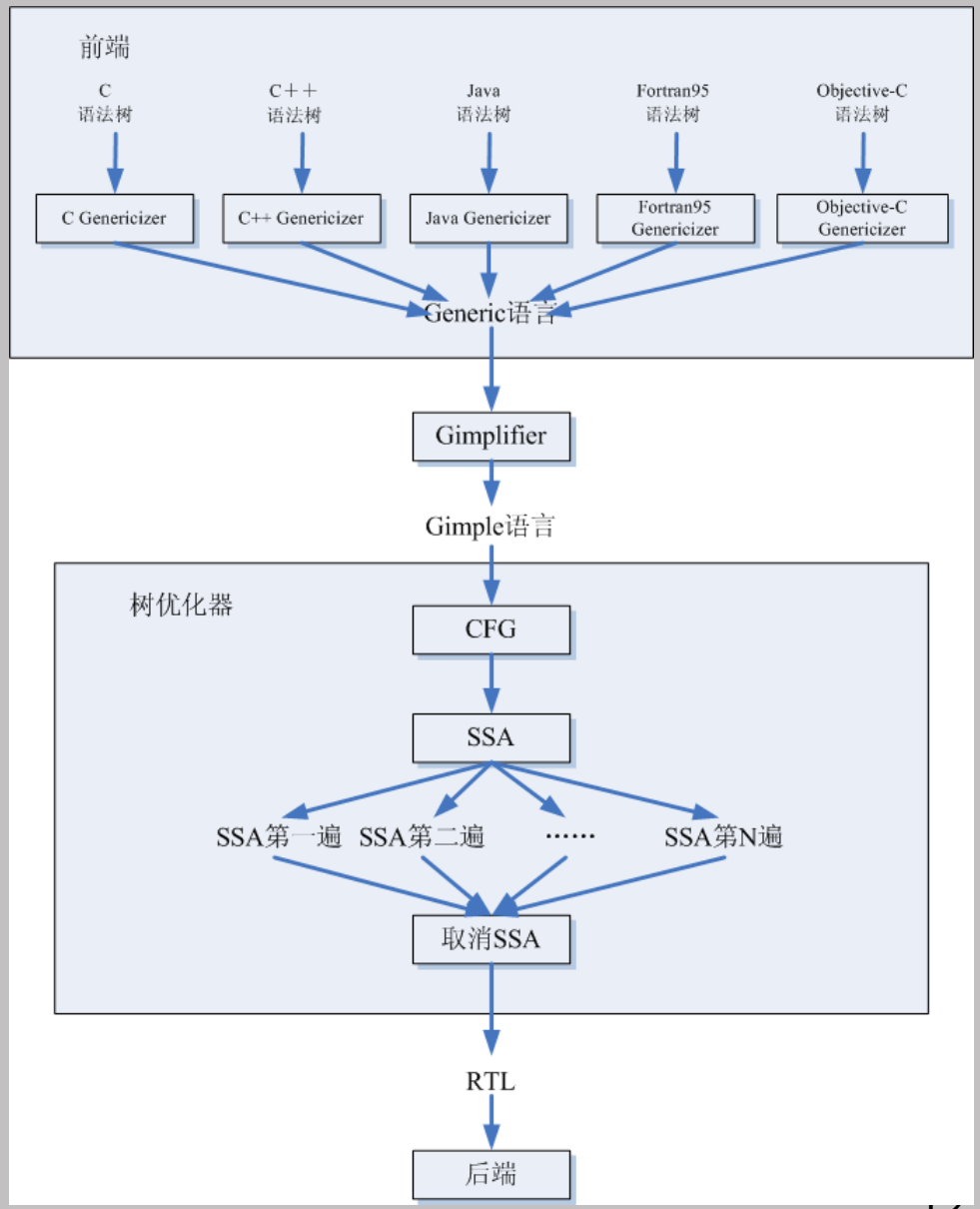
BDL的架构


被检测代码
读取
GCC4
规则文件
读取
GCC4
中间代码
BDL前端
BDL后端
输入符号对
BDL
自动机
注册
BDL
规则代码
BDL
内建函数
匹配函数
动作函数
停机函数
回朔函数
.....
BDL
中间代码
结果输出

GCC3概况


前端
C
C 解析器
C++
C++ 解析器
Java
Java 解析器
Fortran95
Fortran95
解析器
Objective-C
Objective-C
解析器
RTL
后端
RTL
优化器
代码
生成器
目标代码

GCC4概况


前端
C
语法树
C++
语法树
Java
语法树
Fortran95
语法树
Objective-C
语法树
C Genericizer
C++ Genericizer
Java Genericizer
Fortran95
Genericizer
Objective-C
Genericizer
Generic语言
Gimplifier
Gimple语言
树优化器
CFG
SSA
SSA第一遍
SSA第二遍
……
SSA第N遍
取消SSA
RTL
后端

GCC4的新特性

- 中间代码与前端无关，有可能同时查找多种语言的Bug
- Generic和Gimple的出现，简化了前端的复杂情况
- CFG位于Gimple层，可以被很方便的使用

使用GCC的优点

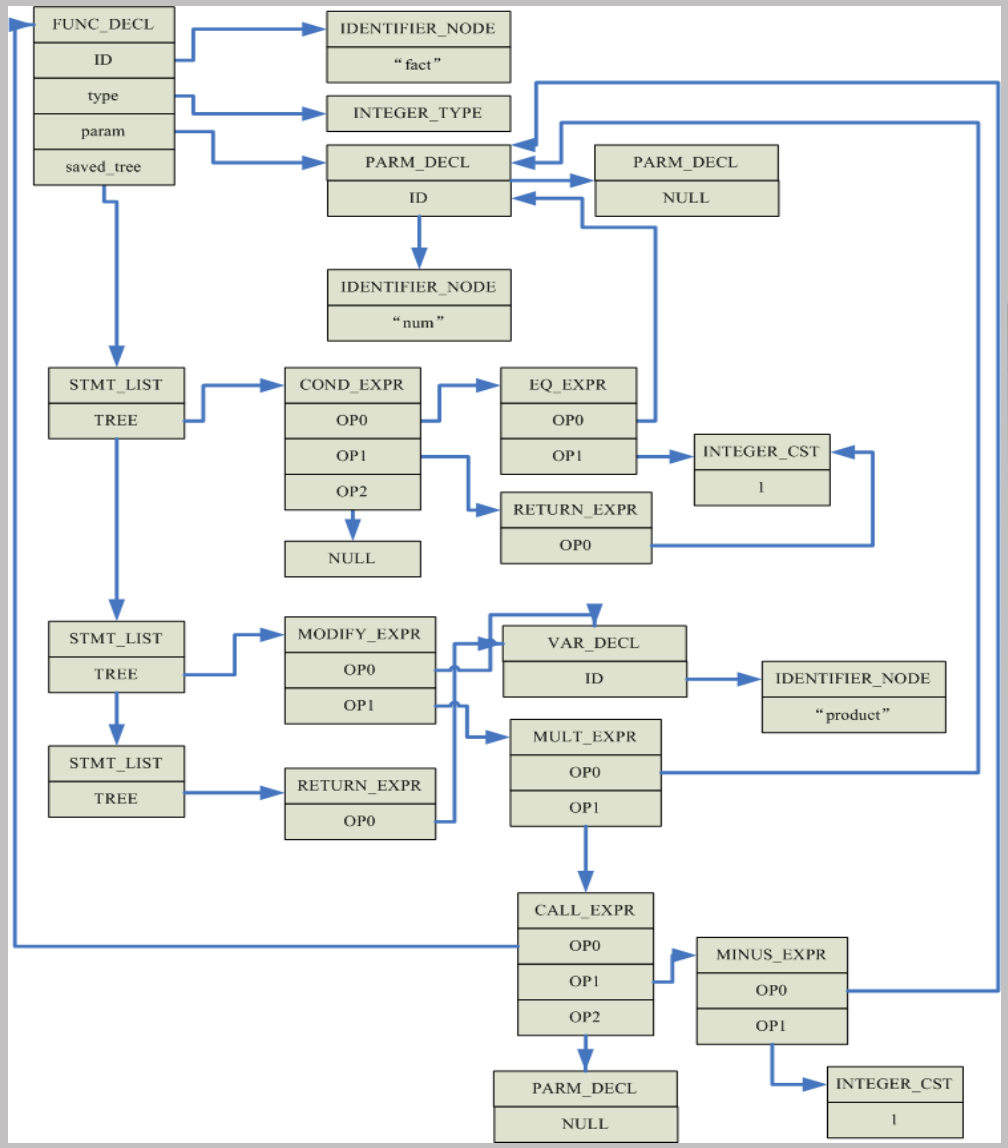
- 写一个符合工业标准的C前端是非常耗费时间的 。使用GCC可以不必在词法分析、语法分析、中间代码和CFG生成上花费过多的精力
- GCC4 由于Gimple和Generic的出现，使得中间代码既能保持足够多的源代码信息，又能生成CFG图
- GCC4本身带有很多工具函数
- 现在大部分的软件都依靠Makefile机制完成编译
- 由于Gimple是中间语言，语前端语言类型无关的。相同的BDL描述，有可能检测出不同语言的同种缺陷

使用GCC的缺点

- GCC4是一个很大的框架，各模块之间的耦合度较大，因此很难按需要对它作大规模的修改
- GCC4中间代码的内容过为丰富，我们需要的只是很小的一部分，因此使用GCC4会占用较多的内存资源较多
- 访问GCC4中间代码时，如果内存访问处理不当，容易导致编译器的崩溃

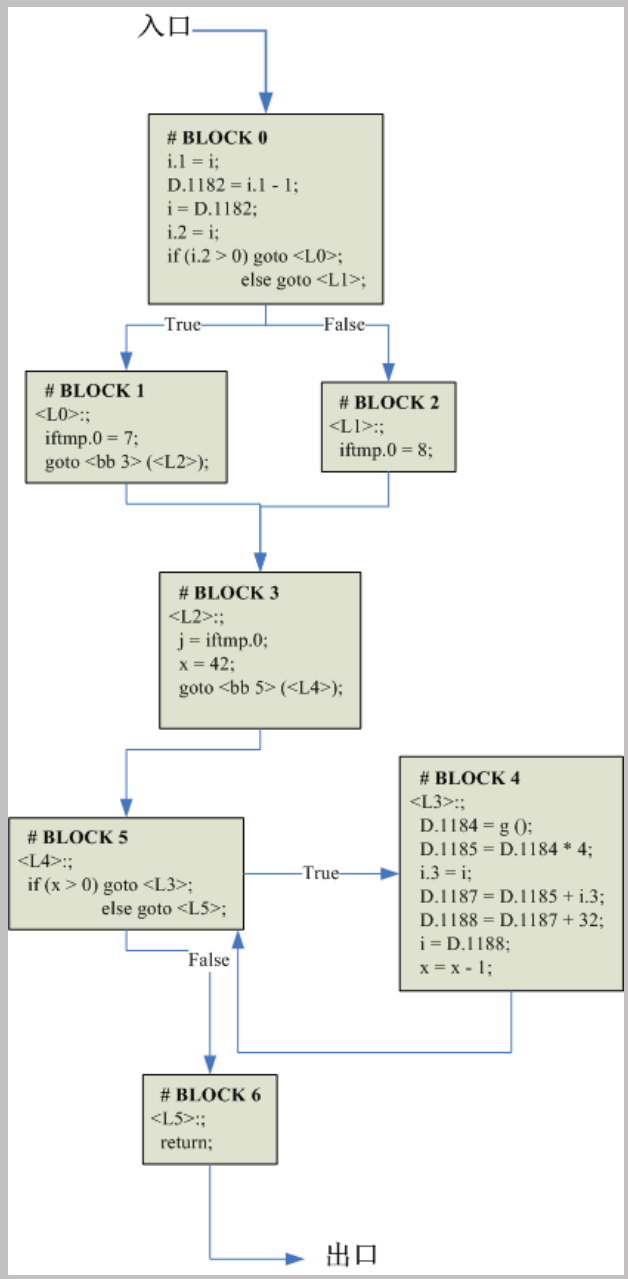
GCC4的语法树与数据结构

```
unsigned int fact
(unsigned int num)
{
        unsigned int product;
        if(num = = 1)
            return 1;
        product = num *
                    fact(num – 1);
        return product;
}
```



GCC4的Gimple和CFG

```
int i;
int g(void);
void foo(void)
{
    int j = (--i, i > 0? 7: 8);
    for(int x = 42; x > 0; --x)
        i += g() * 4 + 32;
}
```



BDL的前端

- BDL前端解析器负责从指定文件中读取并分析缺陷的描述
- 尽量满足C编程习惯
- 使用现成的前端开发工具加快开发速度
  - 使用Flex进行词法分析,Bison进行语法分析
  - 具体的词法语法规则简单
- 关键字：sm / option / declare / declare_sv / ID标示符 / “=>” / 类型关键字(any char int scalar ptr) / END

BDL举例 - Spinlock

```
sm spin_lock_checker{                    // 规则名称spin_lock_checker
    option sensitive;                    // 生成路径遍历信息
    option 5;                            // 最多5次循环
    declare_sv { any } SMIntLock;        // 声明状态机变量

    SMIntLock.Init: { spin_lock(SMIntLock) }
    =>   SMIntLock.Locked { msg("init==>locked\n"); }
    SMIntLock.Locked: { spin_lock(SMIntLock) }
    =>   SMIntLock.Error { err("double locked.\n"); backtrace(); }
    SMIntLock.Locked: { spin_unlock(SMIntLock) }
    =>   SMIntLock.Init { msg("locked==>init\n"); }
    SMIntLock.Init: { spin_unlock(SMIntLock) }
    =>   SMIntLock.Error { err("lock unlocked var.\n");
    backtrace(); }
    SMIntLock.Locked: { END }
    =>   SMIntLock.Error { err("lock hold when leave.\n");
    backtrace(); }
}
```

测试结果/driver/net/slip.c Linux 2.3.50

```
>>>>>>>>>>> function : sl_ioctl <<<<<<<<<<<<<<<<<<<
init==>locked
lock hold when leave.
======== Backtrace start ========
No.1: var sl->lock(file:slip.c, line 1255) Init => Locked
======== Backtrace stop =========

init==>locked
locked==>init
lock unlocked var.
======== Backtrace start ========
No.1: var sl->lock(file:slip.c, line 1255) Init => Locked
No.2: var sl->lock(file:slip.c, line 1275) Locked => Init
No.3: var sl->lock(file:slip.c, line 1313) Init => Error
======== Backtrace stop =========
```

错误定位

```
1248  static int sl_ioctl(struct net_device *dev,struct ifreq *rq,int
    cmd){
......
1255  spin_lock_bh(&sl->lock);                    // ← 加锁
......
1262  switch(cmd){
1263      case SIOCSKEEPALIVE:
1264      /* max for unchar */
1265          if (((unsigned int)((unsigned long)rq->ifr_data)) > 255)
1266              return -EINVAL;                 // ← 忘记解锁
......
1282      case SIOCSOUTFILL:
1283          if (((unsigned)((unsigned long)rq->ifr_data)) > 255)
/* max for unchar */
1284              return -EINVAL;                 // ← 忘记解锁
```

错误定位（续）

```
1248  static int sl_ioctl(struct net_device *dev,struct ifreq
    *rq,int cmd){
        ......
1255  spin_lock_bh(&sl->lock);                    //  ←加锁
        ......
1262  switch(cmd){
1263    case SIOCSKEEPALIVE:
        ......
1275        spin_unlock_bh(&sl->lock);  // ←解锁
1276        break;
        ......
1312  };
1313  spin_unlock_bh(&sl->lock);                  // ←解锁
1314  return 0;
1315 }
```

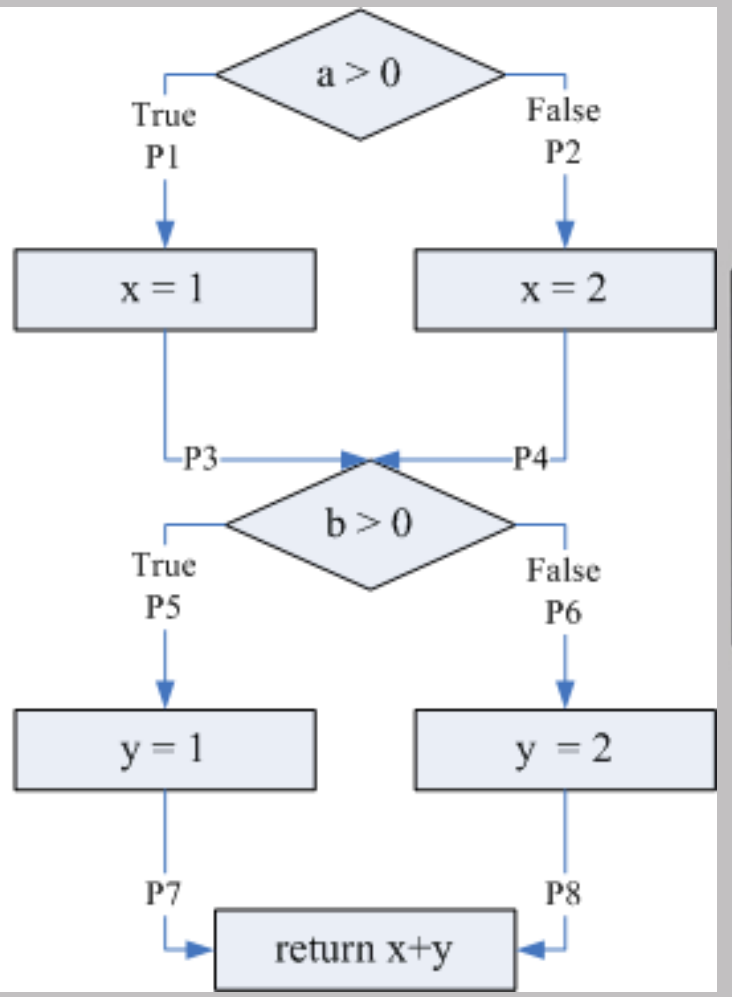
BDL后端 — 路径遍历

- 指查找从函数入口到函数出口所有可能的执行路径(Execution Path)
- 采用DFS算法
- 右图的所有路径
  P1→P3→P5→P7
  P2→P4→P5→P7
  P1→P3→P6→P8
  P2→P4→P6→P8
- 循环次数可控

```
int foo ( int a, int b )
{
    int x, y;

    if ( a > 0 )    x = 1;
    else            x = 2;
    if ( b > 0 )    y = 1;
    else            y = 2;

    return x+y;
}
```

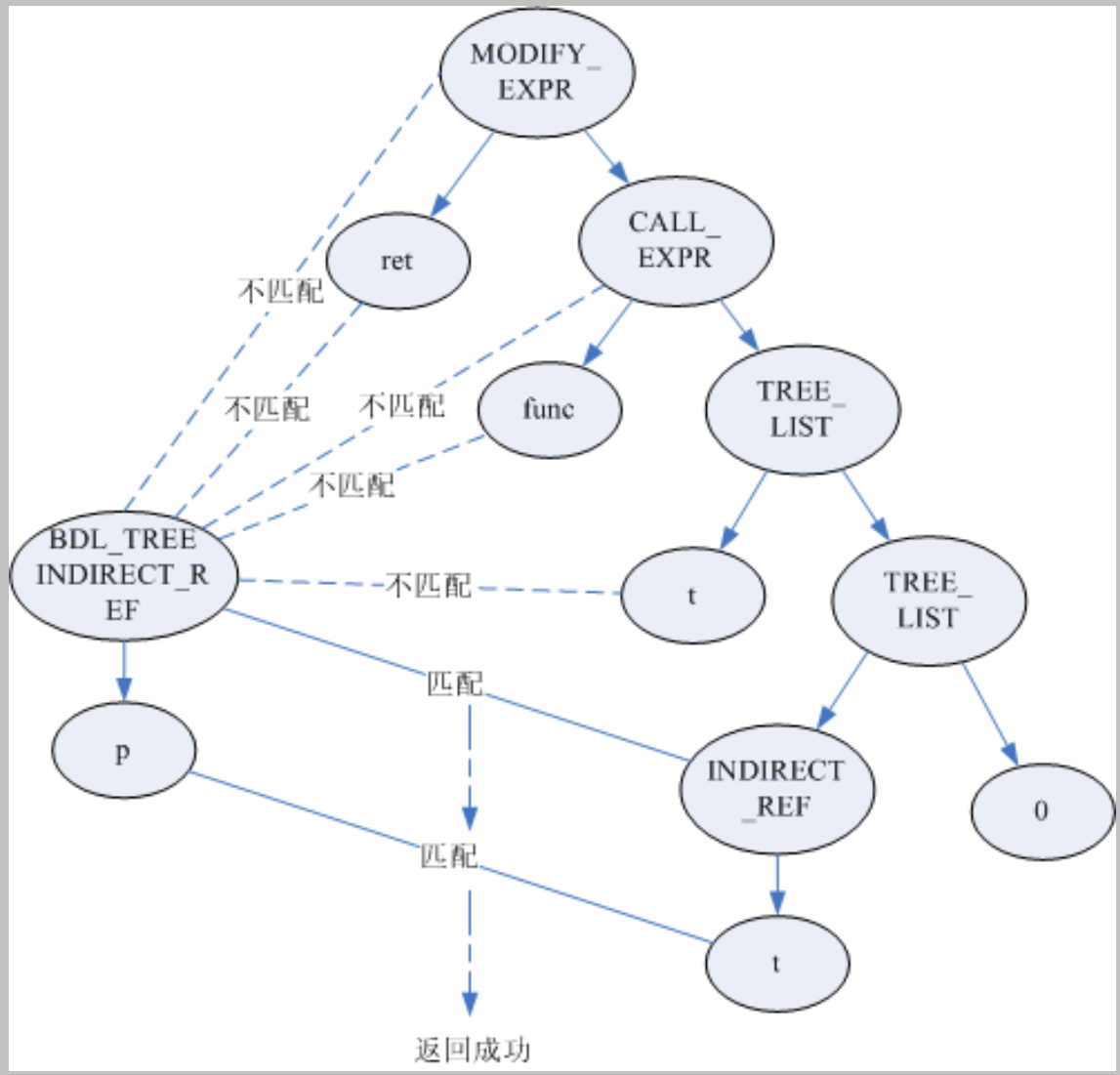


BDL后端 — 变量拆分

- 当匹配模式为func(target, any, any)
  如何区分
  - func(p, NULL, NULL)
  - func(NULL, p, NULL)
  - func(NULL, NULL, p)
- 后端需要将func(a,b,c)的变量区别开来，分三次输入自动机，即(a, func), (b, func), (c, func)
- GCC的中间代码可以区别不同范围的同名标示符

BDL内建函数

- 匹配函数
  - 使用递归匹配
  - 单独处理函数调用
  - *p 匹配
    ret = func (t, *t, 0)
- 动作函数
- 停机函数
- 回溯函数



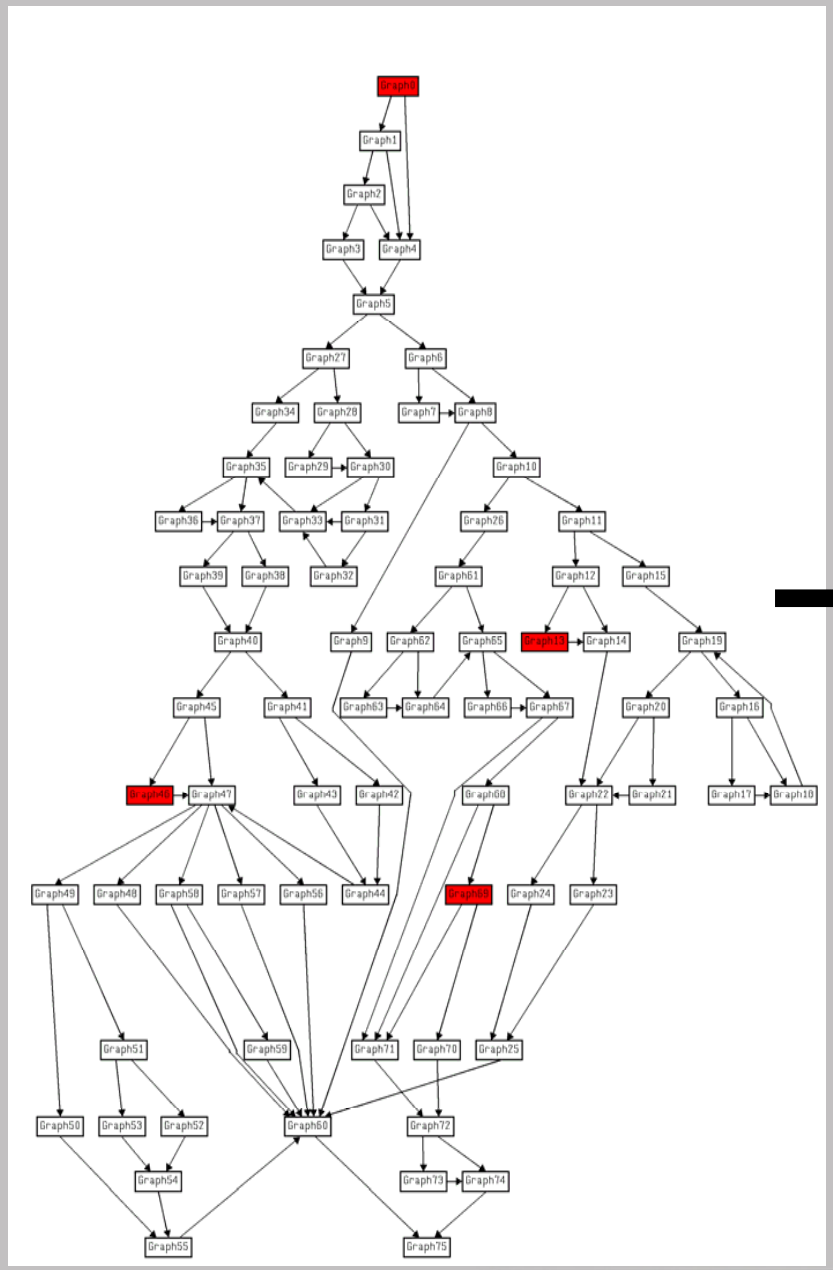
BDL自动机引擎

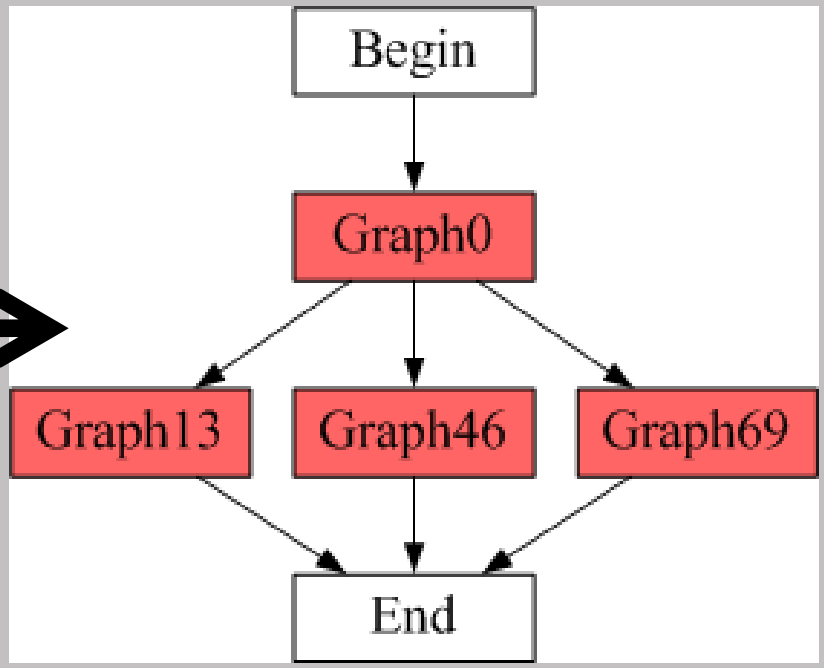
- 为每个变量生成自动机实例
- 从BDL前端获取状态转换规则列表
- 从后端获取目标变量和目标语句的输入流
- 调用内建匹配函数判断状态是否转移
- 自动机停机时，调用内建停机函数输出出错信息

改进 — 路径数缩减

- 程序逻辑非常复杂的时候，BB的数量很多,生成的路径成千上万,大部分都不是需要关心的
- 对于CFG中的每一个BB节点 ,使用amodule_try_symbol 判断是否可以触发状态转变,将可转变的BB标记出来
- 对于可触发的状态机变量,如果是右值,则递归将左值记录下来,并将该语句所在的BB标记出来
- 对于未标记的BB删除,将该BB的前驱和后继连接起来，合并相同的边，同时注意保留if的真假分枝
- Linux 2.6.16 JFS的txLock，缩减之前BB为76，路径数为4104，按照lock规则缩减之后BB为4，路径数为3

缩减举例





改进 — 全局分析

- 前文的内容是过程内分析，在实际的代码检查中，往往还需要进行过程间分析
- gcc以单个文件为编译单元，每编译完一个单元就会释放所有的语法树，从而无法完成对多个源文件的过程间分析。
- 思路：

（1）直接利用cc1。

（2）将GCC产生的所有中间代码导出来，再重构语法树和控制流图。

改进 — 缓冲与注释支持

- 路径缩减的对象是BB，但是对于同一BB内部，仍然有优化余地
- 对于一个BB和一个变量状态机实例sti，如果进入BB前sti的状态为s_x，离开BB后的状态变为s_y，那么<sti, s_x, s_y>是唯一确定的，即sti和s_x能够唯一的确定s_y
- 对每一个BB设置一个缓冲，当发生状态转换时，将<sti, s_x, s_y>保存到缓冲中，下次如果访问有sti, s_x ，则直接读取，不许调用状态机进行转换
- 无法将“//”和“/* */”表示的注释信息保留在GCC4的中间代码中
  - 利用__attribute__关键字,添加对添加“bdl”的支持

还可以做什么？

- 能否和某些动态检测技术相结合？
- 能否改善指针的分析机制？
- 检测脚本存在的漏洞？
- 向商业产品借鉴？

Thanks!