北大软件 | 打造富有生命力的软件
CREATE THE VITALITY OF SOFTWARE

北京大学 PEKING UNIVERSITY

基于静态分析的程序缺陷自动检测与自动修复技术

马森 高庆

北京北大软件工程股份有限公司

2017年7月

内容提要

- 引言
- 静态程序分析技术
- 基于静态分析技术的缺陷检测
- 基于静态分析技术的缺陷修复
- 总结与展望

背景及必要性

1978年，Hatford体育场倒塌

7000万美元

1985年，Therac-25放射治疗仪

三人死亡，三人重伤

1988年，蠕虫程序

蔓延互联网

1996年，Ariane火箭

损失5亿美元

2005年，EDS 儿童资助系统

损失约5亿多英镑

2006年，熊猫烧香

损失约1亿美元

软件灾难

OWASP历年安全问题Top10多数与**程序代码**相关

OWASP Top 10 (**2004**)
A1 – 未验证输入
A2 – 失效的访问控制
A3 – 失效的身份认证和会话管理
A4 – 跨站脚本 (XSS)
A5 – 缓冲区溢出
A6 – 注入漏洞
A7 – 不正确的错误处理
A8 – 不安全的加密存储
A9 – 拒绝服务
A10 – 不安全的配置管理

OWASP Top 10 (**2007**)
A1 – 跨站脚本 (XSS)
A2 – 注入漏洞
A3 – 恶意文件执行
A4 – 不安全的对象直接引用
A5 – 跨站请求伪造 (CSRF)
A6 – 信息泄露和不正确错误处理
A7 – 失效的身份认证和会话管理
A8 – 不安全的加密存储
A9 – 不安全的通信
A10 – 没有限制URL访问

OWASP Top 10 (**2010**)
A1 – 注入漏洞
A2 – 跨站脚本 (XSS)
A3 – 失效的身份认证和会话管理
A4 – 不安全的对象直接引用
A5 – 跨站请求伪造 (CSRF)
A6 – 安全配置错误
A7 – 不安全的加密存储
A8 – 没有限制URL访问
A9 – 传输层保护不足
A10 – 未验证的重定向和转发

OWASP Top 10 (**2013**)
A1 – 注入漏洞
A2 – 失效的身份认证和会话管理
A3 – 跨站脚本 (XSS)
A4 – 不安全的对象直接引用
A5 – 安全配置错误
A6 – 敏感信息泄露
A7 – 功能级访问控制缺失
A8 – 跨站请求伪造 (CSRF)
A9 – 使用含有已知漏洞的构件
A10 – 未验证的重定向和转发

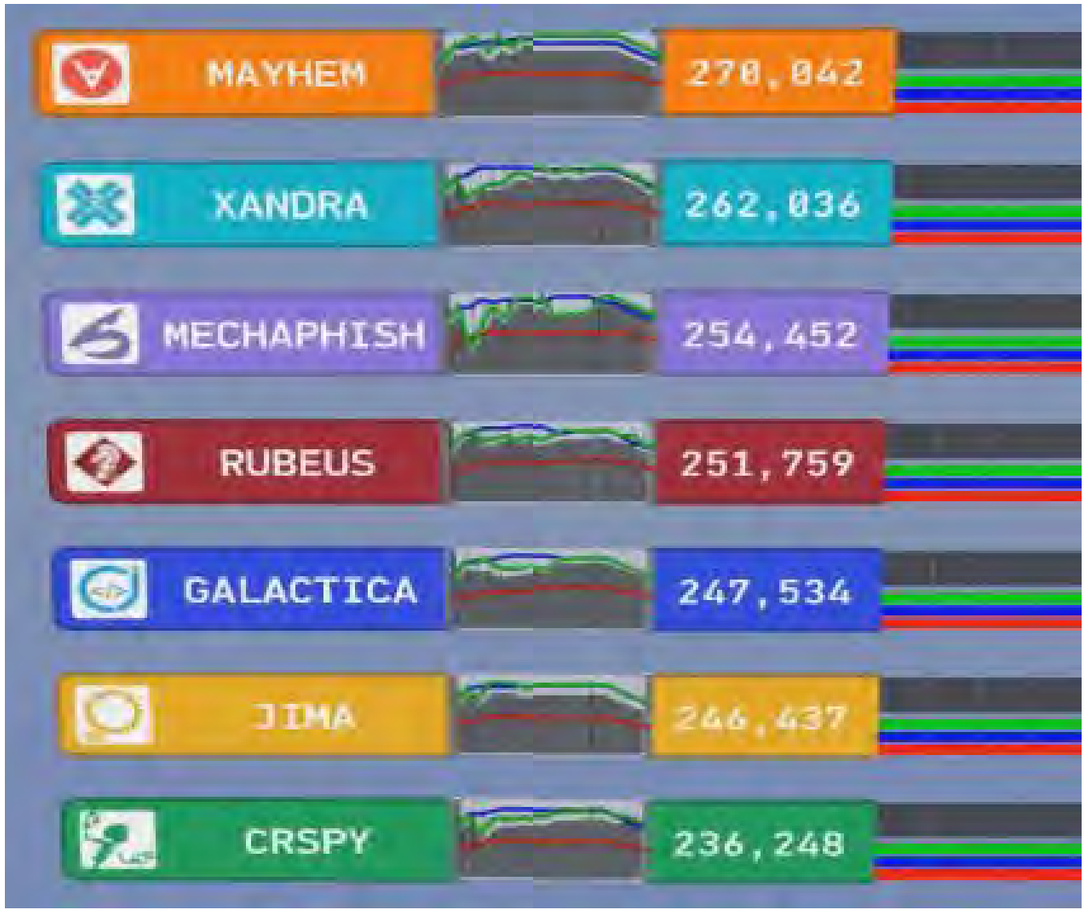
只有代码中的安全缺陷得以及时消除，最终形成的软件产品才能具备较高的安全性，有效降低整体系统安全风险

代码静态分析可以帮助用户从根源上减少30%-70%的软件安全问题

代码静态分析工具主要被国外产品所垄断，国内尚无成熟工具，存在安全隐患，无法做到自主可控

自动化漏洞发现比赛-CGC

CGC决赛
七支参赛队伍
每支队伍：
1280核
16TB内存
128TB存储空间



- 2016年8月美国DAPRA举办首届CGC挑战赛，目的是自动挖掘、利用及修复代码中的安全漏洞。
- 卡耐基梅隆大学Brumley团队的Mayhem夺得第一名
- MayHem在和人类的PK中处于劣势

在漏洞挖掘领域，机器远未像Alpha Go在智能搜索领域取得的成绩显著

自动化漏洞发现比赛-CGC

1st:
CMU
Mayhem

动态模块
污点分析
插桩代码
虚拟化执行

静态模块
符号执行
路径选择
利用生成
状态保存

2nd:
GrammaTech & Virginia Tech
Xandra

静态漏洞分析 ↔ 动态符号执行

静态漏洞分析 → 漏洞确认引擎

动态符号执行 → 漏洞确认引擎

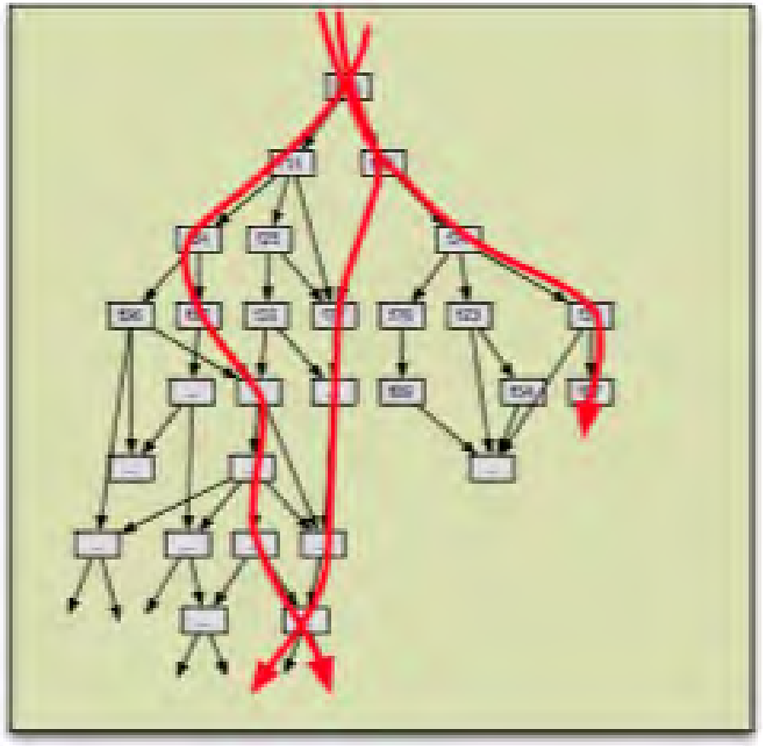
3rd:
UC-Santa Barbara
MechaPhish

模糊测试 → 动态分析

符号执行 ↔ 动态分析

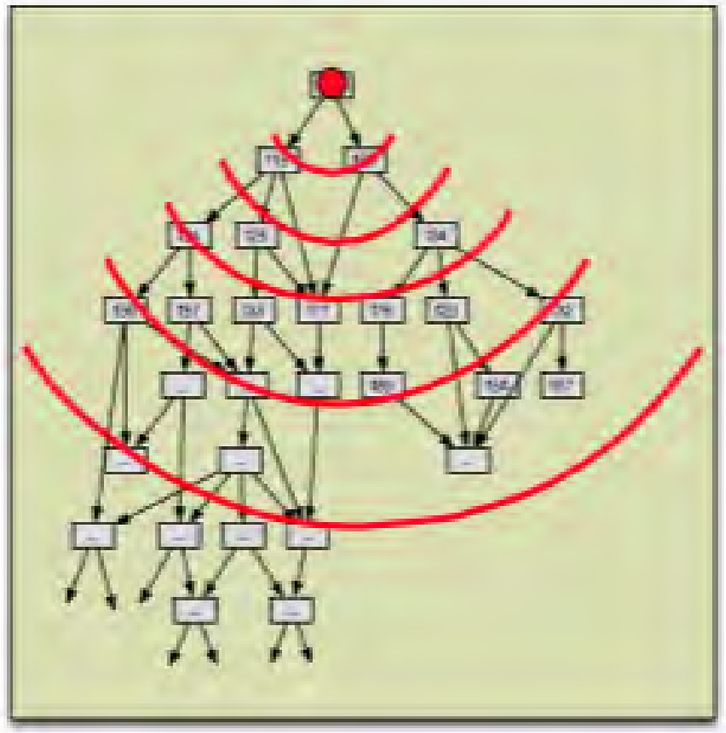
静态语义检测与动态测试的互补

动态测试



- 执行单一的路径
- 对关键的功能性测试很有效
- 对于Corner情况很难

静态语义检测

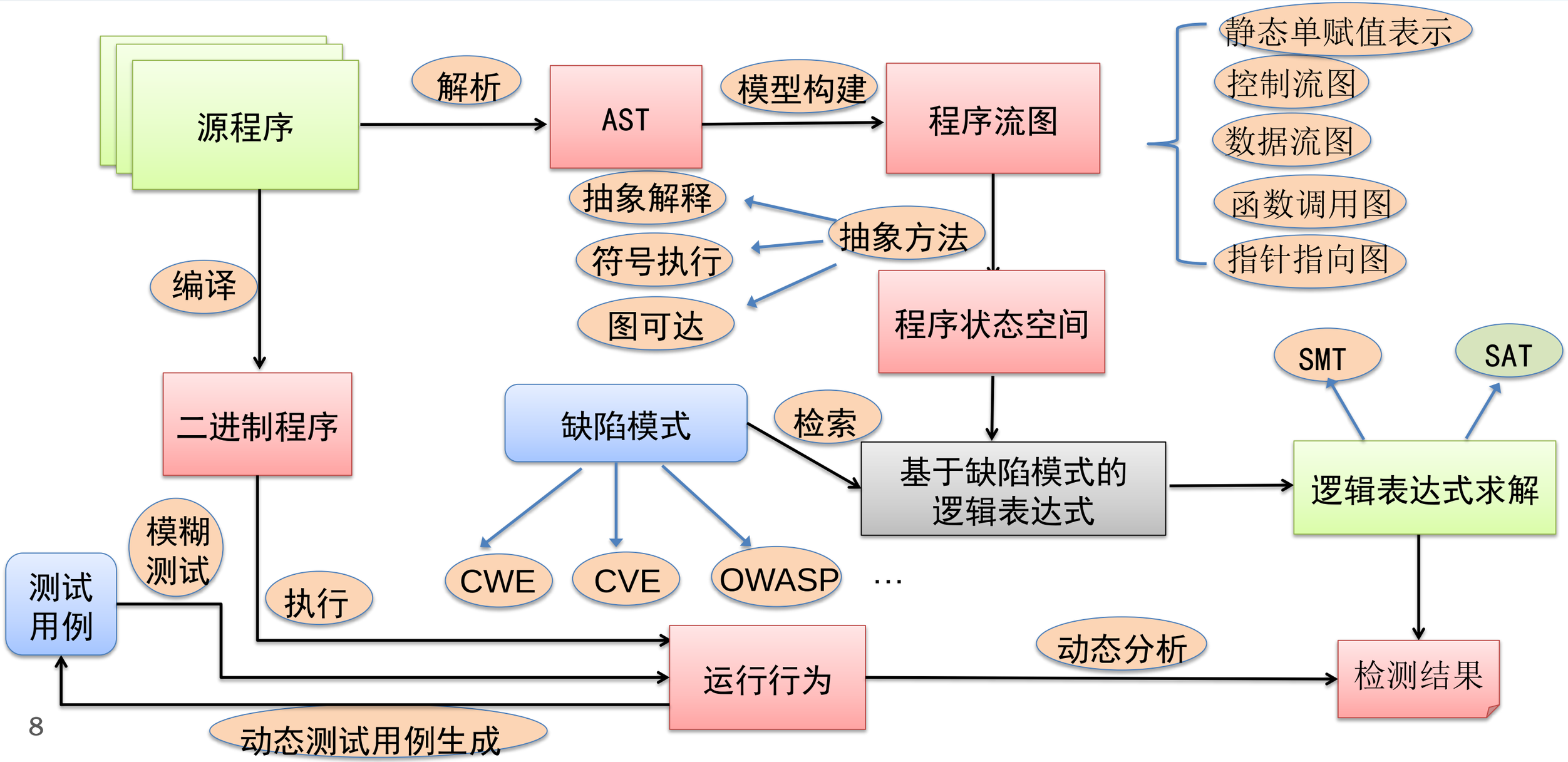


- 执行“符号”分析
- 对于具有缺陷模式的Corner情况非常有效
- 不能进行功能性测试

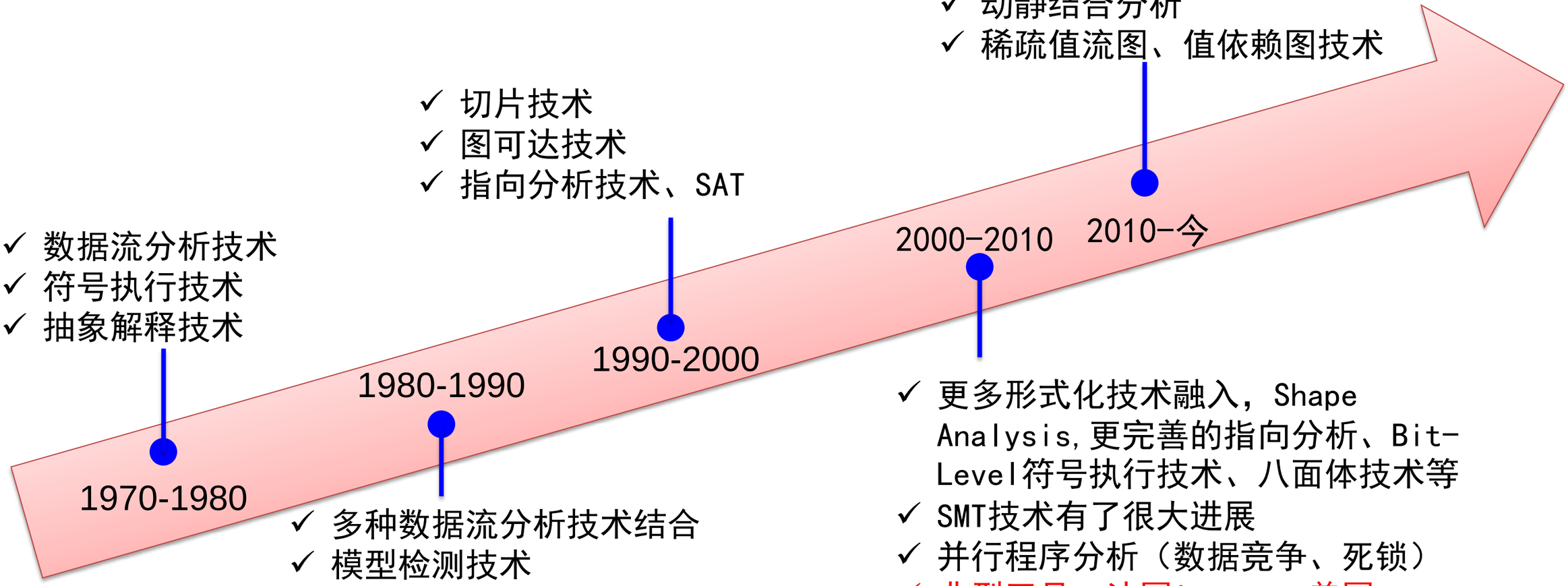
内容提要

- 引言
- 静态程序分析技术
- 基于静态分析技术的缺陷检测
- 基于静态分析技术的缺陷修复
- 总结与展望

未知缺陷自动发现的基本流程



未知缺陷静态检测技术发展历程



未知缺陷自动发现的基本流程中相关概念

基本分析相关

Control Flow Graph（CFG）：控制流图

Call Graph（CFG）：函数调用图

SSA:静态单赋值

Super Graph：控制流图与调用图的

Mod-effect：修改影响分析，表达指针修改后对其他引用的影响

Dataflow Analysis：包括前后支配分析、到达定值分析、活跃变量分析等

继承关系分析：分析类之间的继承关系

Points-to：指向分析，表达程序中指针的指向关系

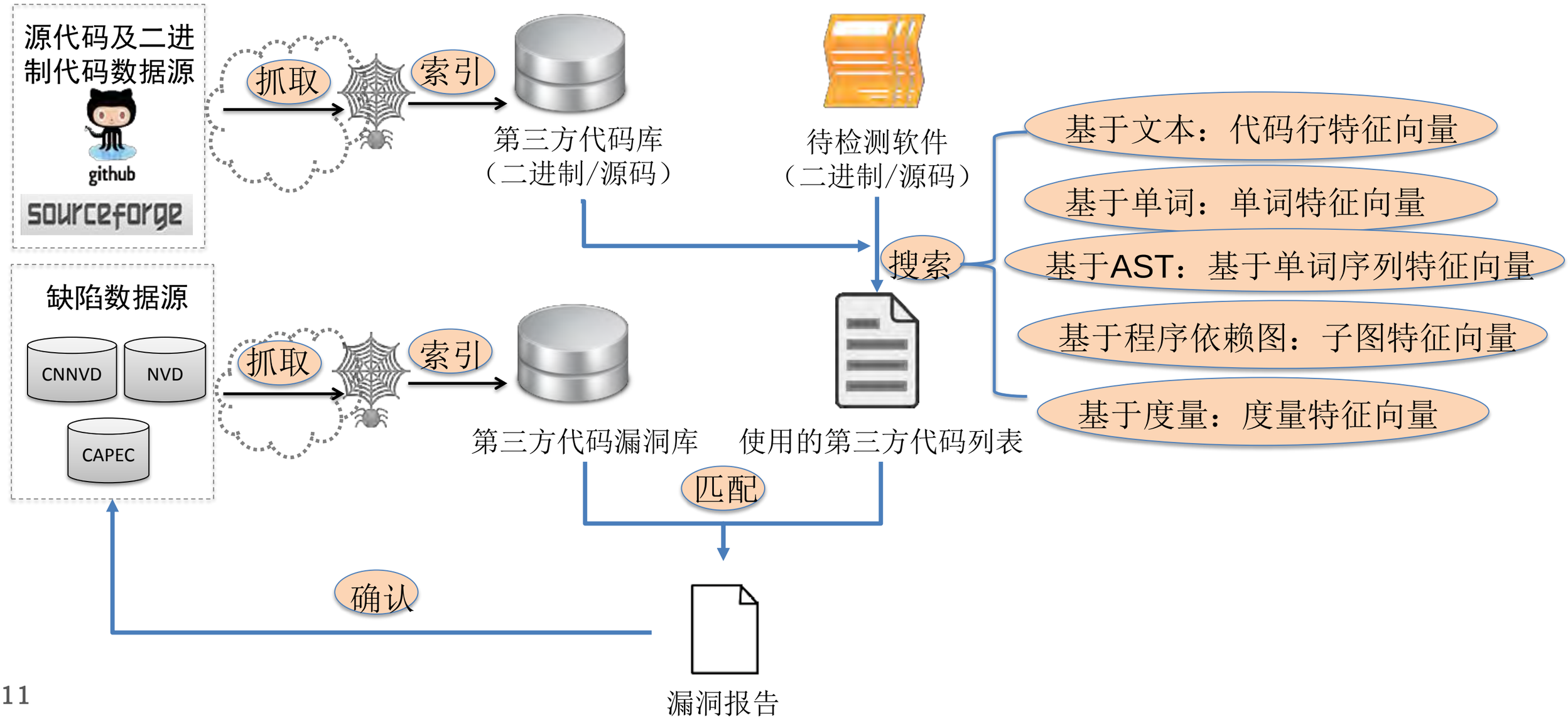
建立程序状态空间

- **抽象解释理论**是使用一个基于“抽象对象域”上的较低代价的计算过程来抽象逼近基于“受检程序指称对象域”上的计算过程，从而使得程序抽象执行的结果能够尽量反映出程序真实运行过程中的部分信息

符号执行是将程序源代码中变量的值采用抽象化符号的形式表示，模拟程序执行目的是分析程序中变量之间的约束关系，不需制定具体的输入数据，将变量作为代数中的抽象符号处理结合程序的约束条件进行推理，结果是一些描述变量间关系的表达式

图可达性分析根据分析问题的不同将程序应用一些函数分析技术或者算法转换为相应的图形，然后利用上下文无关语言的理论，计算点与点的可达性

基于同源分析的已知缺陷发现的基本流程



挑战一：源码漏洞检测精度与效率的折衷

Coverage（覆盖率）

How thoroughly are all possible execution paths and data values covered by the analysis?

Automation（自动化程度）

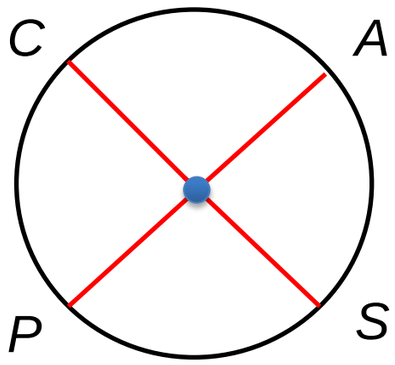
How much manual effort is required?

Precision（准确性）

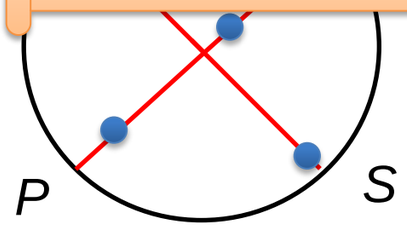
How precisely does the analysis correspond to the actual software that is executed?

Scalability（可伸缩性）

How large of a code base can be analyzed by the technique or tool?



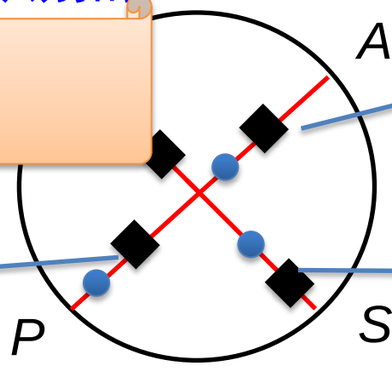
理论中的理想情况



现实的情况

能够接受的漏报，马上能识别的

精度与效率的折衷问题



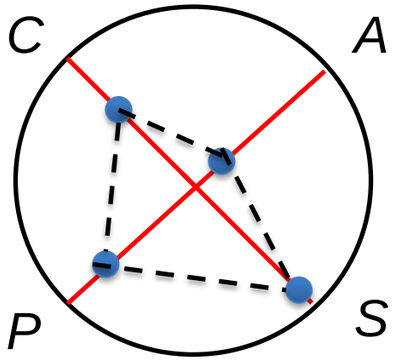
无须自定义规约

误报率30%以下

每分钟1万行以上

现实中能够接受的情况

挑战一：源码漏洞检测精度与效率的折衷



影响折衷问题的因素：

分析采用的算法是否是：

1. 路径敏感的
2. 流敏感的
3. 上下文敏感的
4. 域敏感的

只有做到全部敏感精度才不会有缺失
理论上是不可能的，现实只能做到部分敏感

挑战一：源码漏洞检测精度与效率的折衷

```
template <typename X> class ACE_Auto_Basic_Ptr
{
public:
    typedef X element_type;

    // = Initialization and termination methods
    explicit ACE_Auto_Basic_Ptr (X * p = 0) : p_ (p) {}

    ACE_Auto_Basic_Ptr (ACE_Auto_Basic_Ptr<X> & ap);
    ACE_Auto_Basic_Ptr<X> &operator= (ACE_Auto_Basic_Ptr<X> & rhs);
    ~ACE_Auto_Basic_Ptr (void);

    X *get (void) const;

protected:
    X *p_;
};

template <typename X> class ACE_Auto_Ptr : public ACE_Auto_Basic_Ptr <X>
{
public:
    typedef X element_type;

    // = Initialization and termination methods
    explicit ACE_Auto_Ptr (X * p = 0) : ACE_Auto_Basic_Ptr<X> (p) {}

    X *operator-> () const;
};
template<class X> ACE_INLINE X *
ACE_Auto_Basic_Ptr<X>::get (void) const
{

    return this->p_;
}
template<class X> ACE_INLINE
ACE_Auto_Basic_Ptr<X>::~ACE_Auto_Basic_Ptr (void)
{

    delete this->get ();
}
```

几个研究的事实：

- Context sensitive最优的算法
  Tom Reps 提出的CFL-Reachability的解法 $O(ED^3)$复杂度，但在大型工程不可接受

- 已经证明Field-Sensitive和Context-sensitive不能共存的
  导致面向对象语言漏洞自动检测效果一般

- 单函数的圈复杂度在5000以上，精确的跨函数分析的效率基本很难接受，例如GitHub的项目DosBox,最好的工具也只是低误报

挑战二：逻辑表达式求解的问题

- SAT: use propositional logic as the formalization language
  - + high degree of efficiency
  - - expressive (all NP-complete) but involved encodings

```
void myfree(int *p ,int a){
    if(a > 0){
        free(p);
    }else if(a == 0){
        free(p);
    } else if(a <0){
        free(p);
    }
}
```

```
int main(){
    int a;
    int *p = (int*)malloc(MIN * sizeof(int));
    scaf(&a);
    myfree(p, a);
    return 0;
}
```

SAT:
A\/B\/C =>
Satisfied

SMT:
a>0 \/ a==0 \/ a<0
=> true

- SMT: propositional logic + domain-specific reasoning
  - + improves the expressivity
  - - certain (but acceptable) loss of efficiency

SMT与SAT相比SMT问题具有表达能力更强、抽象层次更高等优点

main函数申请数组后调用myfree()函数SAT分析会存在memory leak误报；SMT分析则不会存在误报。

挑战二：逻辑表达式求解的问题

Z3: 微软开发的目前使用最广稳定性最好，但仅支持Windows平台，且不开源

Yices2: Z3之前使用最广稳定性最好的Solver，由Z3作者加入微软之前撰写，支持所有平台且开源

SMTInterpol: 基于LGPL v3下由java开发的支持函数和在整数与实线性算数的无量词的相结合

Platform		Features					
Name	OS	SMT-LIB	CVC	DIMACS	Built-in theories	API	SMT-COMP
Beaver	Linux,Windows	v1.2	No	No	bitvectors	OCaml	2009
CVC4	Linux,Mac OS,Windows	Yes	Yes		rational and integer linear arithmetic, arrays, tuples, records, inductive data types, bit-vectors, strings, and equality over uninterpreted function symbols	C++	2010
MathSAT	Linux	Yes		Yes	empty theory, linear arithmetic, bitvectors, arrays	C/C++,Python, Java	2010
MiniSmt	Linux	partial v2.0			non-linear arithmetic		2010
OpenSMT	Linux,Mac OS,Windows	partial v2.0		Yes	empty theory, differences, linear arithmetic, bitvectors	C++	2011
SMTInterpol	Linux,Mac OS,Windows	v2.0			uninterpreted functions, linear real arithmetic, and linear integer arithmetic	Java	2012
SMT-RAT	Linux,Mac OS	v2.0	No	No	linear arithmetic, nonlinear arithmetic	C++	2015
STP	Linux,OpenBSD, Windows,Mac OS	partial v2.0	Yes	No	bitvectors, arrays	C,C++,Python, OCaml,Java	2011
veriT	Linux,OS X	partial v2.0			empty theory, rational and integer linear arithmetics, quantifiers, and equality over uninterpreted function symbols	C/C++	2010
Yices	Linux,Mac OS,Windows	v2.0	No	Yes	rational and integer linear arithmetic, bit-vectors, arrays, and equality over uninterpreted function symbols	C	2014
Z3	Linux,Mac OS,Windows,FreeBSD	v2.0		Yes	empty theory, linear arithmetic, nonlinear arithmetic, bitvectors, arrays, datatypes,quantifiers	C/C++,.NET,OCaml,Python,Java	2011
...	...	...	...	...	...	...	...

SMT-LIB为标准的SMT输入格式，几乎所有的SMT Solver都支持它，并且应用于每年的SMT比赛中

挑战二：逻辑表达式求解的问题

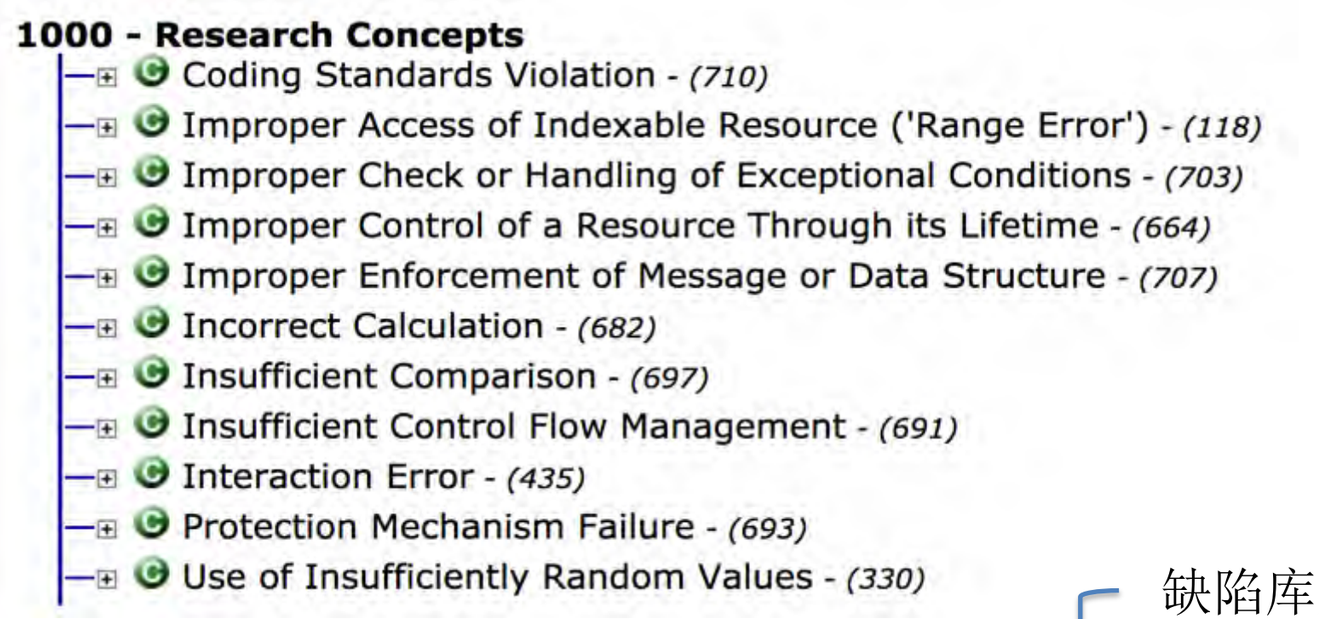
1、千个以上Atom求解效率较低，针对静态检测的优化研究较少

2、针对字符串的处理不完善，导致一些检测效果不好，如SQL注入

- SMT支持字符集有限,当前主流SMT仅支持ASCII码字符集,对中文支持不完善
- SMT支持的字符串操作有限,仅支持极其简单的字符串操作

挑战三：缺陷模式的提取

目前，静态缺陷的缺陷模式主要来自于：Common Weakness Enumeration(CWE)



- 11类1000余种安全模式，包括了能用静态分析检测所有模式类别

- 模式类别全，具体的实例不足
- 缺陷模式提取效率低，不够完善

- 缺陷库人工提取（CVE+NVD）
- 历史修改记录提取
- 最新研究采用网络自动挖掘（PRMiner）

例1：连续拷贝漏报

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  int main()
4.  {
5.          char uri[128];
6.          char service[]="abc";
7.          char realm[128];
8.          int i=0;
9.          for(i=0;i<128;i++)
10.         {
11.                 realm[i]='a';
12.         }
13.         strcpy(uri, service);
14.    strcat(uri, "/");
15.    strcat(uri, realm);
16.    return 0;
17.}
```

uri和realm都是128 uri 中还有service等字符串，可能导致缓冲区溢出。
连续拷贝处理导致的缓冲区溢出缺陷

CVE编号：CVE-2013-0249
出错工程：curl7.28.1

例2：缺少指针偏移信息漏报

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. #define MAXNAME 2
4. int main(int argc,char** argv)
5. {
6.         static char buf[MAXNAME + 1];
7.         char *p=buf;
8.         int x=100;
9.         while(x-->0)
10.        {
11.                p++;
12.        }
13.        *p='\0';
14.        return 0;
15.}
```

因为缺少对指针偏移信息的描述，由此导致漏报

CVE编号：CVE-2002-1337
出错工程：sendmail8.12.7

例3：循环相关的漏报

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  char a[3]={'a','b','c'};
4.  int main()
5.  {
6.       int id=0;
7.       int i=0;
8.      for(i=0;i<4;i++)
9.     {
10.         char x=getchar();
11.         printf("%d\n",id);
12.        if(x == a[id]) {//id可以取到3，取到3的时候越界
13.        id++;
14.    }
15. }
16. return 0;
17.}
```

因为循环导致的区间信息计算不准确，导致误报

CVE编号：CVE-2014-8962

出错工程：libflac 1.3.0

例4：区间信息计算不准确的漏报

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #define BUFSIZE (1024*1024）
4.  int main()
5.  {
6.              char buf[BUFSIZE];
7.              char head[5];
8.              char * request;
9.              scanf("%s",request);
10.             int len=strlen(request);
11.         int f=1;
12.             if (len > BUFSIZE + sizeof(head))
13.             {
14.                 printf("Request too big!");
15.                 return 0;
16.             }
17.             read(f, buf, len);
18.             return 0;
19. }
```

+变成-区间信息才正确

CVE编号：CVE-2014-2892
出错工程：libmms 0.6.3
100个工程编号：107

例5：复杂的别名关系导致的漏报

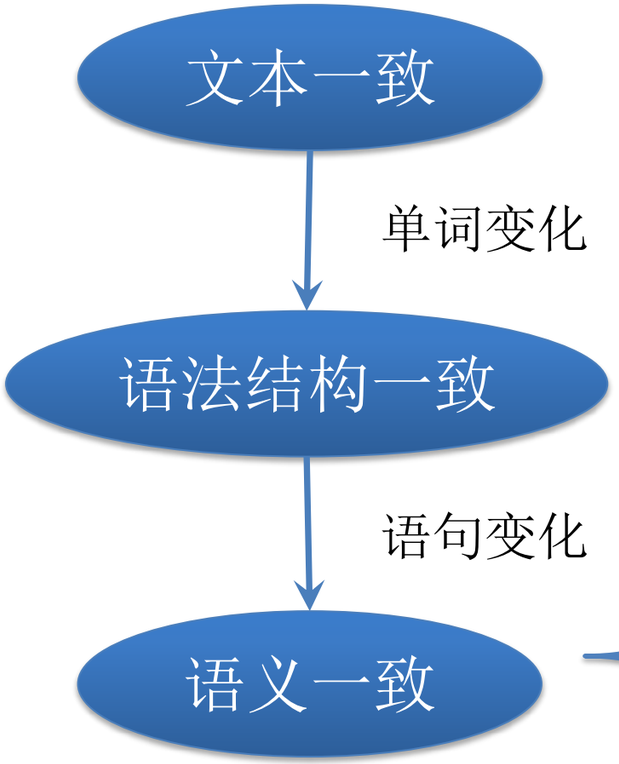
```
void comprexx( char **fileptr);
int main(int argc,char **argv){
     char   **filelist，  **fileptr;
     filelist = fileptr = (char **)malloc(argc*sizeof(char *));
      *filelist = null;
     for (argc--, argv++; argc > 0; argc--, argv++)
    {
          *fileptr++ = *argv;//污染源
          *fileptr = NULL;
     }
     if (*filelist != NULL)
     {
           for (fileptr = filelist; *fileptr; fileptr++)//别名
                comprexx(fileptr);
     }
   return 0;
}
void comprexx( char **fileptr)
{
    char        tempname[100];
    strcpy(tempname,*fileptr);//缓冲区溢出
}
```

循环导致的复杂的别名指针关系导致的漏报

CVE编号：CVE-2001-1413
出错工程：ncompress 4.2.4
100个工程编号：无

挑战四：同源分析

- 大规模数据集上的高精度匹配
- 语义级别同源分析



高效的索引算法
- 索引规模：选取代表性单词
- 特征向量：包含尽可能丰富的程序信息

高效的查询算法
- 粒度：基于文本/单词
- 查询方式：基于特征向量

基于二进制汇编代码检测

结合动态方法
- 黑盒：比较输入输出
- 白盒：分析运行行为

内容提要

- 引言
- 静态程序分析技术
- 基于静态分析技术的缺陷检测
- 基于静态分析技术的缺陷修复
- 总结与展望

源代码漏洞自动检测工具

工具名称	研发机构	主要分析技术
Coverity Prevent	美国Synopsis公司	符号执行，模式匹配
Klocwork Insight	美国Rogue Wave Software公司	符号执行，区间分析
HP Fortify	美国HP公司	数据流分析，模式匹配
FindBugs	美国Maryland大学	模式匹配
Clang	美国UIUC	符号执行
cpplint	美国谷歌	模式匹配
Polyspace	美国MathWorks公司	抽象解释
Astree	法国巴黎LIENS实验室	抽象解释
Sparrow	韩国首尔国立大学	抽象解释

典型工具

国内工具：

- 北京大学 CoBOT
  - 基于值依赖图
  - 支持C/C++/Java语言
- 北京邮电大学 DTS
  - 基于符号执行的实现方式
  - 支持C/C++/Java语言

国外工具

- Synopsys Coverity
  - 基于符号执行的实现方式
  - 支持C/C++/Java/C#/Python语言

国内工具要么支持缺陷种类少，要么误漏报过高运行效率低

二进制漏洞自动检测工具

工具名称	研发机构	主要分析技术
Veracode	美国Veracode公司	禁运未知
Codesonar	美国GrammaTech公司	禁运未知
Vine	美国加州大学伯克利分校	数据流分析，符号执行
BAP	美国卡耐基梅隆大学	数据流分析，符号执行
Codesurfer/x86	美国威斯康辛大学	值集分析
McVeto	美国威斯康辛大学	模型检查
Bindead	德国慕尼黑理工大学	抽象解释
Angr	美国加州大学圣芭芭拉分校	符号执行
Jakstab	德国达姆施塔特理工大学	符号执行
Valgrind	开源社区（起源于英国剑桥）	代码插桩，污点分析
TEMU	美国加州大学伯克利分校	模拟执行，污点分析
Triton	法国Quarkslab	动态符号执行
Protecode	美国Synopsys公司	模式匹配

典型工具

- Bitblaze
  - 动静结合，基于Vine和TEMU
  - 效率较低，不支持某些指令集的解析
- Triton
  - 基于动态符号执行的测试用例生成
  - 难以应用于大型工程
- Protecode
  - 基于同源分析检测二进制代码漏洞
  - 不支持未知漏洞的检测

二进制漏洞检测工具在国内处于空白状态

缺陷检测工具应用：编码规则、缺陷与漏洞测试

国际标准

- **MISRA C/C++**
  - ✓ 由MISRA提出的C语言开发标准。目的在于提高嵌入式系统的安全性及可移植性
  - ✓ MISRA C 2004，分为21类，共计141项
  - ✓ ISO 17961 安全编码标准，45类
  - ✓ MISRA C++ 2008，分为21类，共计228项

国家标准

- **GJB 8114-2013**
  - ✓ 软件C/C++语言安全子集及编程规范
- **GJB 5369-2005**
  - ✓ 由航天科工集团公司提出，并结合航天型号软件特点，经裁减补充而成。分为15类，共计138项

行业标准

- **921 C-2007**
  - ✓ 依据GJB 5369，制定了航天“921”工程C语言软件编程准则和要求。分为15类，共计163项
- **BACC C-2008**
  - ✓ 航天某单位C语言编程规范，分为17类，共计134项
- **CRSC C-2014**
  - ✓ 高铁领域C语言编程规范，分为18类，共计218项

指针缺陷类型\安全等级	Critical (1级)	Severe (2级)	Error (3级)
内存泄漏		4	
资源泄漏		1	
空指针解引用	16		
释放非堆内存	1	1	
释放未分配的内存	2		
使用已释放的内存	4	4	
函数返回局部变量的地址	3		
未初始化使用	4	1	
释放不成功			1
合计	30	11	1

其他缺陷类型\安全等级	Critical (1级)	Severe (2级)	Error (3级)
不可达路径	2	1	
非法计算			4
库函数(打印)			5
库函数(网络访问)		4	20
库函数(输入输出)		10	22
库函数(本地访问)		1	6
库函数(宏)			4
合计	2	16	61

CWE缺陷

- OWASP中活跃且严重性等级较高的漏洞：
  - ✓ 缓冲区溢出（27种）
  - ✓ 数组越界 （24种）
  - ✓ 整数溢出（15种）
  - ✓ 浮点数溢出（15种）
  - ✓ SQL注入（10种）

CoBOT(库博)缺陷检测工具的优势

安全可控

- 自主知识产权-更加完善的支持国产标准（唯一支持GJB 8114检测），能够支持标准定制（CRSC, BACC）
- 全面支持国产化环境（中标麒麟、WPS等）

能力强

- 编译不通过情况下检测
- 与国外工具相媲美的检测精度（误漏报率）和效率，部分指标国际领先
- B/S模式，减少部署工作量

功能强

- 能够支持编码规则、语义缺陷、安全漏洞以及自动用例生成代码覆盖率检测

持续服务

- 规则，缺陷，报告模板量身定制
- 专业的服务团队，专业的软件质量咨询服务

CoBOT（库博）缺陷检测工具的检测能力

① 编码规则

- 支持MISRA C 2004 、 MISRA C++ 2008、 GJB 5369-2005 、 GJB 8114-2013 、 GB/T

② 程序缺陷

- 支持CWE缺陷中严重等级较高的100余种缺陷类型
  - ✓ 资源（内存泄漏）
  - ✓ 缓冲区（数组越界）溢出
  - ✓ 非法计算
  - ✓ 整数溢出
  - ✓ 释放非堆内存
  - ✓ 空指针解引用
  - ✓ 死代码
  - ✓ 使用释放后内存
  - ✓ 差一缺陷
  - ✓ 未初始化使用

⑤ 编译环境&操作系统

- 编译环境
  - ✓ 支持GCC、CC、VC06、VC08、CCS、Workbench、Tornado等
- 操作系统
  - ✓ 中标麒麟、Windows、Linux（Ubuntu）、Solaris（X86、Sparc）、AIX等

③

- 基于OWASP，支持其中涉及生产重性等级较高的八类90种安全漏洞检测
  - ✓ 输入验证与表示(29种)
  - ✓ API滥用(10种)
  - ✓ 安全功能部件(10种)
  - ✓ 时间与状态(8种)
  - ✓ 错误处理(4种)
  - ✓ 代码质量(9种)
  - ✓ 封装(11种)
  - ✓ 环境(9种)

④ 软件度量

- 支持20种关于文件、类以及函数/方法的代码度量，主要用于分析代码的清晰性、可维护性以及可测试性
  - ✓ 圈复杂度
  - ✓ 代码注释率
  - ✓ 循环数
  - ✓ 函数扇入/扇出

缺陷检测工具测试结果

- 为了验证编码规则和程序缺陷检测方面的效果，需要进行三种类型的测试

中等规模开源项目，测试误报率和漏报率

选择SPEC 2000（由标准性能评价组织SPEC开发的用于评测通用型CPU性能的基准程序测试组）中十个开源项目进行验证分析，项目规模在万行至几十万行之间，主要目标是测试COBOT的误报率和漏报率

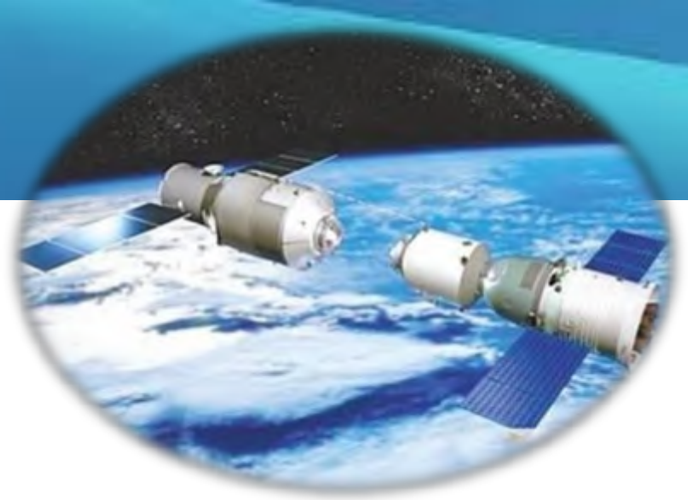
较大规模开源项目，测试运行效率及误报率

选择规模在三个百万行以上的大型开源项目，主要目标是测试工具的运行效率及误报率

实际应用项目，综合评估实际运行效果

将工具部署到实际运行环境中，对约数十万行代码进行检测，综合评估实际运行效果。

航天某单位应用比较：程序缺陷分析



项目		CoBOT（库博）		Klocwork		相同缺陷			
名称	代码行	检测时间	误报数/总数	检测时间	误报数/总数	合计	资源泄漏	空指针解引用	使用释放内存
1	4108	20.2	0/4	48.1	0/2	2	2	0	0
2	3715	21.1	2/5	44.4	0/3	3	3	0	0
3	1287	10.5	0/3	38.2	0/3	3	0	3	0
4	331	10.8	0/0	28.1	0/0	0	0	0	0
5	3641	13.5	0/0	63.2	0/0	0	0	0	0
6	2744	14.8	0/0	61.3	0/0	0	0	0	0
7	785	10.8	0/0	31.3	0/0	0	0	0	0
8	3304	21.5	0/1	51.4	0/0	0	0	0	0
9	7339	16.1	0/0	21.2	0/0	0	0	0	0
10	14480	20.7	0/0	31.6	0/0	0	0	0	0
11	8515	17.2	0/2	42.6	0/0	0	0	0	0
12	2417	20.7	0/6	49.2	0/6	6	6	0	0
13	2669	21.9	0/8	45.3	0/8	8	8	0	0
14	3457	32.2	1/9	56.4	0/8	8	8	0	0
15	2173	22.6	0/6	50.3	0/6	6	6	0	0
16	4158	20.5	0/0	45.2	0/0	0	0	0	0
17	3251	23.1	0/0	43.5	0/0	0	0	0	0
18	1650	13.3	0/2	65.3	0/2	2	2	0	0
19	1912	11.1	0/3	43.2	0/3	3	3	0	0
20	1235	13.2	0/0	35.5	0/0	0	0	0	0
21	1621	10.2	1/1	65.5	0/0	0	0	0	0
22	537	10.8	0/0	35.4	0/0	0	0	0	0
23	14254	20.3	0/1	42.8	0/0	0	0	0	0
总计	89583	397.1	4/51	1039	0/41	41	38	3	0

CoBOT（库博）发现了6个Klocwork未发现的缺陷：

- 项目23：空指针解引用 1个；
- 项目11：资源泄漏 2个；
- 项目8：资源泄漏 1个；
- 项目1：资源泄漏 2个

漏报率：CoBOT在10%左右，Klocwork在20%以上
误报率：CoBOT在10%-20%之间，Klocwork低于10%

轨道某单位应用比较：编码规则检测

CoBOT		Testbed	
漏报率	误报率	漏报率	误报率
1.32%	0.06%	2.60%	0.32%

误报率：相比于CoBOT，Testbed误报率高出0. 26%，漏报率高出1. 28%。可见，误漏报率均高于CoBOT

ID	描述	Testbed漏报	Testbed误报	CoBOT漏报	CoBOT误报
37S	过程参数只有类型没有标识符	2	0	0	0
110S	使用单行注释//	34	0	0	0
130S	使用了不允许使用的 Included 文件	7	0	0	0
331S	字面值需要一个 u 后缀	14	0	67	0
58S	存在空语句	18	0	0	4
59S	if 语句中没有 else 分支	100	0	0	0
62D	指针参数应被定义为常量	74	1	6	0
67S	在模块中使用#define	38	0	0	0
68S	使用#undef	2	0	0	0
74S	避免使用联合	2	0	0	0
87S	用指针进行代数运算	16	0	0	0
119S	使用嵌套注释	0	2	0	0
21S	……	……	……	……	……
总计		323	40	164	7

航空某单位应用比较：编码规则检测

- 采用GJB 5369编码规则，对襟缝翼系统（代码行数为22051行）进行了比对测试
- 比对对象是Testbed Ver7.8.3

Testbed测试较差的条目：

编号	描述	Testbed误报率	Testbed漏报率	COBOT误报率	COBOT漏报率
1.2.4	在结构体定义中使用位域	100.00%	100.00%	0.00%	0.00%
14.1.1	禁止对浮点类型的变量做相等比较操作	100.00%	100.00%	0.00%	0.00%
8.1.3	用八进制数	100.00%	100.00%	0.00%	0.00%
1.1.16	对变量重命名	100.00%	100.00%	0.00%	0.00%
2.1.4	逻辑上关联的表达式需要括号	100.00%	100.00%	0.00%	0.00%
10.2.1	注释中可能包含代码	35.56%	57.35%	1.45%	0.00%
6.1.12	对有符号类型使用位操作	0.00%	98.33%	53.13%	1.64%
6.1.9	符号型/无符号型之间没有使用强制类型转换	0.00%	51.60%	0.00%	0.00%

CoBOT		Testbed	
漏报率	误报率	漏报率	误报率
1.93%	6.06%	18.91%	25.01%

误报率：相比于CoBOT，Testbed误报率高出18.95%，漏报率高出16.98%。可见，误漏报率均远高于CoBOT

值依赖分析技术

值依赖分析技术是一种符号计算技术，通过值依赖分析构建的值依赖模型：

- 表达了程序中全部元素，并建立了的6类值依赖关系及11种值依赖关系

Dependency	Node	Added Edges
Flow	n: $y = x$ $n_w(x)$: each node that writes x	$n_w(x) \rightarrow_x n$
Store	n: $*x = y$ $n_u(x)$: each node that uses x directly, such as y=x or y=f(x)	$n \mapsto_x n_u(x)$
	n: each node that contains &x $n_w(x)$: each node that writes x	$n_w(x) \mapsto_{\&x} n$
Load	n: each node that contains $*x$ $n_w(x)$: each node that writes x	$n_w(x) \dashrightarrow_{*x} n$
	n: each node that contains &x $n_u(x)$: each node that uses x directly, such as y=x or y=f(x)	$n \dashrightarrow_x n_u(x)$
Call	n: $r = f(\dots a_k \dots)$ $n_f(a_k)$: formal parameter n_{ap_k}: actual parameter	$n_f(a_k) \hookrightarrow_f n_{ap_k}$
Return	$r = f(\dots a_k \dots)$ n_{ret}: return node n_r: the node that writes r	$n_{ret} \hookrightarrow_f n_r$
Entry	If a guard exists from entry to definition nodes n_d	$Entry \mapsto_x n_d$

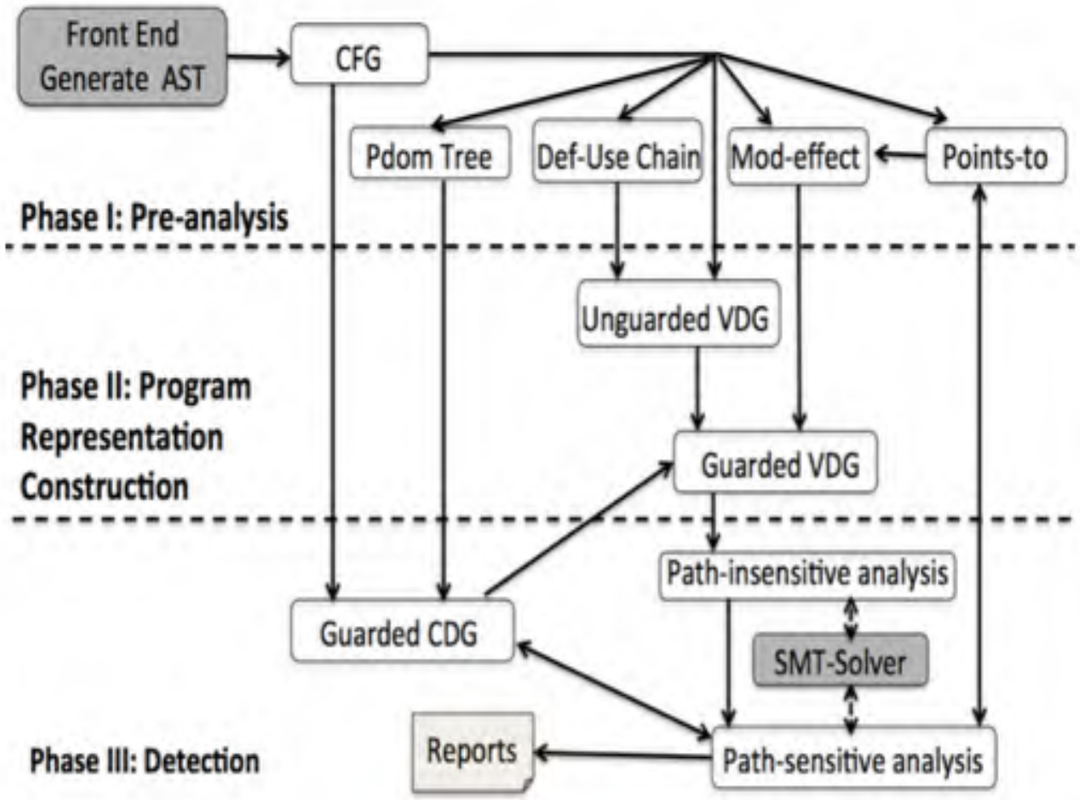
- 综合利用内存的指向信息、变量的可能取值范围信息、增强了模型对程序值依赖描述的准确性
- 通过遍历CFG并利用SSA信息，计算了变量值之间的约束关系，作为值依赖的守卫表达式

值依赖图是一个有向图，由以下四元组组成：

$$\langle N_d^x, \partial, \varphi, N_s \rangle$$

- 其中 N_d^x 表达了X的定义点集合
- ∂ 表达了定义点X到其依赖点的映射及值依赖关系见左图，即 $N_d^x \Rightarrow N_u^x$.
- φ 表达了依赖关系所对应的守卫表达式即 $N_d^x \overset{\varphi}{\Rightarrow} N_u^x$,
- N_s 表达了 φ 所对应的选择语句集合

值依赖分析技术



相对以符号执行为主的解决方案：

1、检测模型更加简化，以变量值的角度构建模型而不是以控制流的方式，这样可以省去与值变化无关的点，使分析效率更高

2、对于精度相对于符号执行，其只能按照符号执行的逻辑不断计算有可能发生状态爆炸，而值依赖计算可以根据程序的复杂度设定迭代次数，是一种主动的折衷方案

3 值依赖模型在由于是跨函数的没有摘要的步骤，所以对于跨函数的缺陷模式精度更高，函数内精度相仿

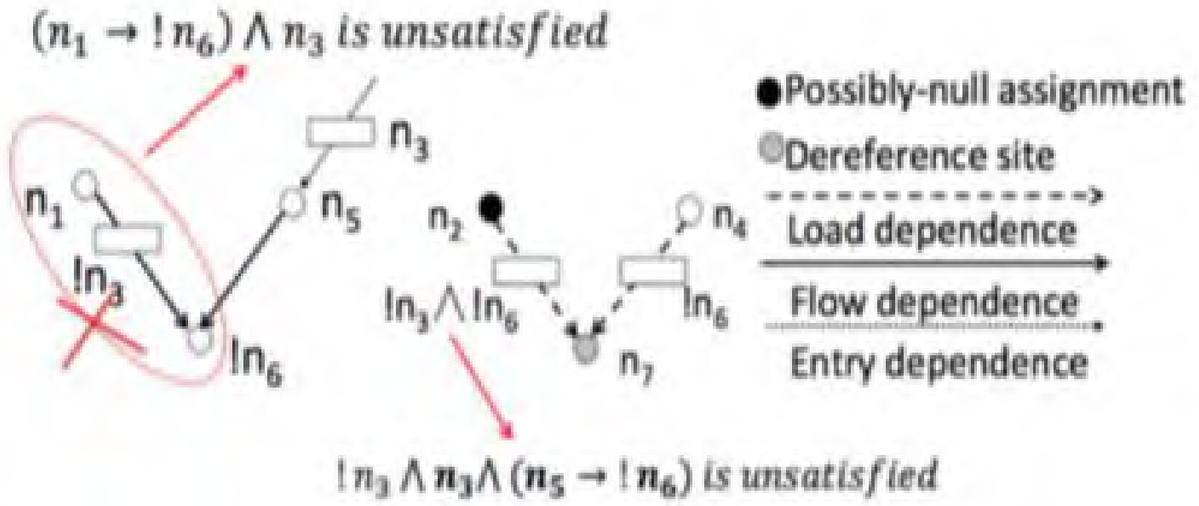
值依赖模型与约束求解：空指针解引用

➢空指针解引用的实例

典型的Infeasible Path程序

```
1   int lastAtomIsMoov = 0;                    //n1
2   MP4Atom* pLastAtom = NULL;                 //n2
3   ...
4   if(*pFirstAtom > 0){                       //n3
5     pLastAtom = &pAtom;                      //n4
6     lastAtomIsMoov = 2;                      //n5
7   }
8   ...
9   if(lastAtomIsMoov < 2){                    //n6
10  ...
11  }
12  else{
13    SetPosition(pLastAtom->getEnd());        //n7
14  }
```

值依赖模型及一次迭代求解过程

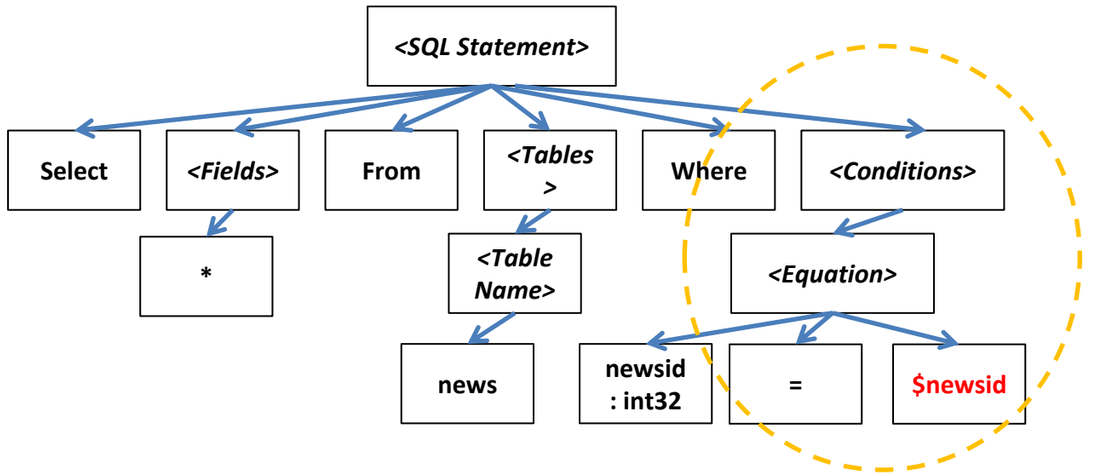


经过一次迭代，约束表达式由原来的$!n_3 \wedge !n_6$转换成了$!n_3 \wedge n_3 \wedge (n_5 \rightarrow n_6)$，得到了Unsatisfied结果，从而消除了误报

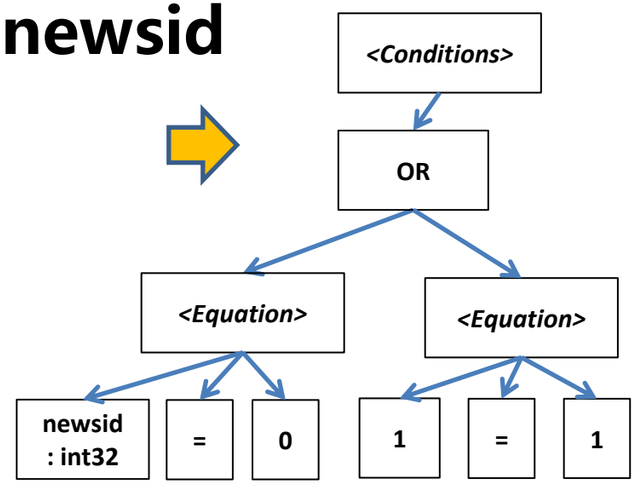
字符串约束求解：SQL注入类漏洞

➢字符串约束求解实例

```
1    $username = ‘posted_user’;
2    $newsid = $_POST[‘posted_newsid’];
     ……
6    if (!preg_match(‘/[\d]+$/’, $newsid )) {
7          Handle exception…
8          exit;
9      }
     ……
16   $newsid = $DB->query("SELECT * FROM ‘news’
                 WHERE newsid=".$newsid);
```

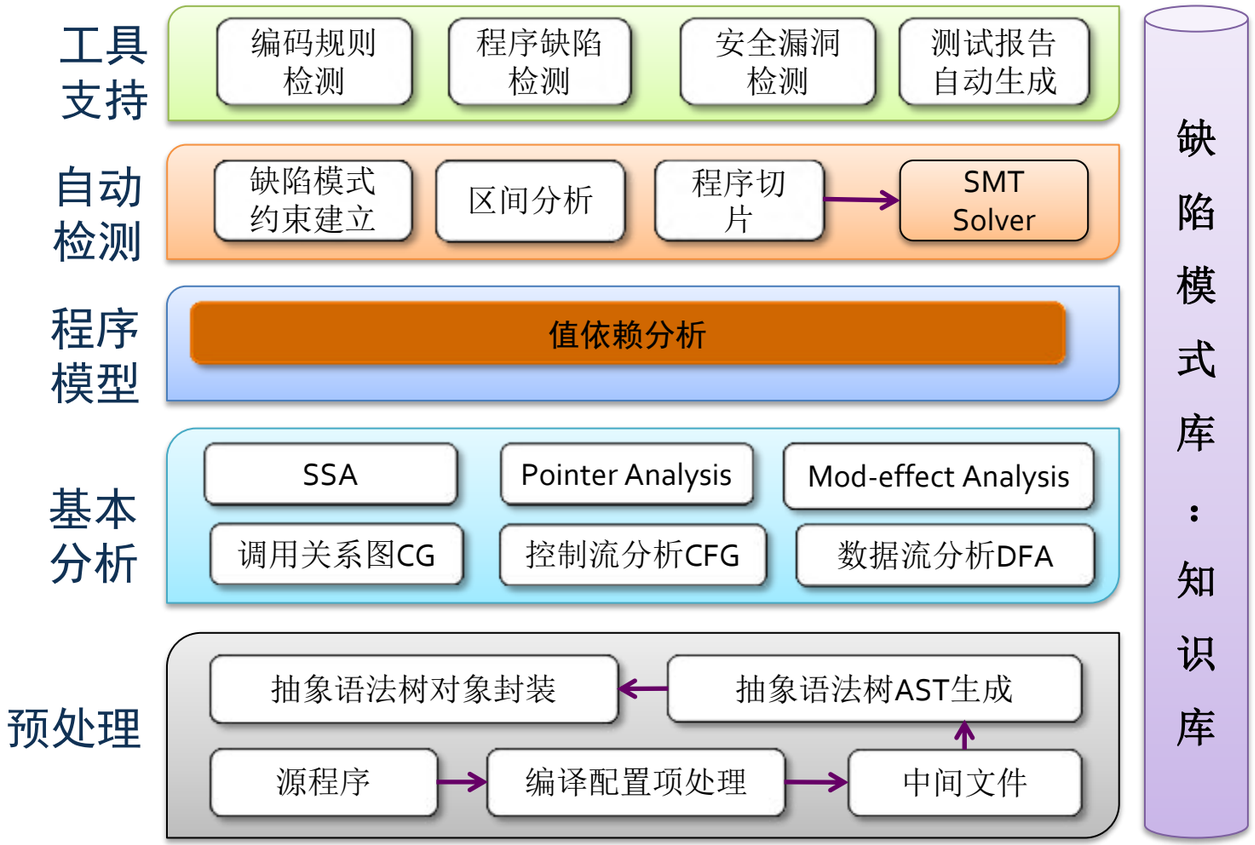


自动生成符号变量$newsid
SQL注入攻击向量
0 or 1=1;#0



分析Web应用源代码漏洞，自动生成攻击向量

基于值依赖分析技术的静态分析工具：库博(COBOT)



- COBOT是一款代码静态分析工具
  - ✓ 采用基于专利技术分析引擎开发的、具有自主知识产权程序静态分析框架，综合运用了多种先进的静态分析技术
  - ✓ 及时发现代码问题，自动识别程序缺陷、安全漏洞和编码风格问题
  - ✓ 并可以进行软件度量及规则的定制化分析、质量报表的模板定制

编码规则检测（10类标准，1000+条规则）

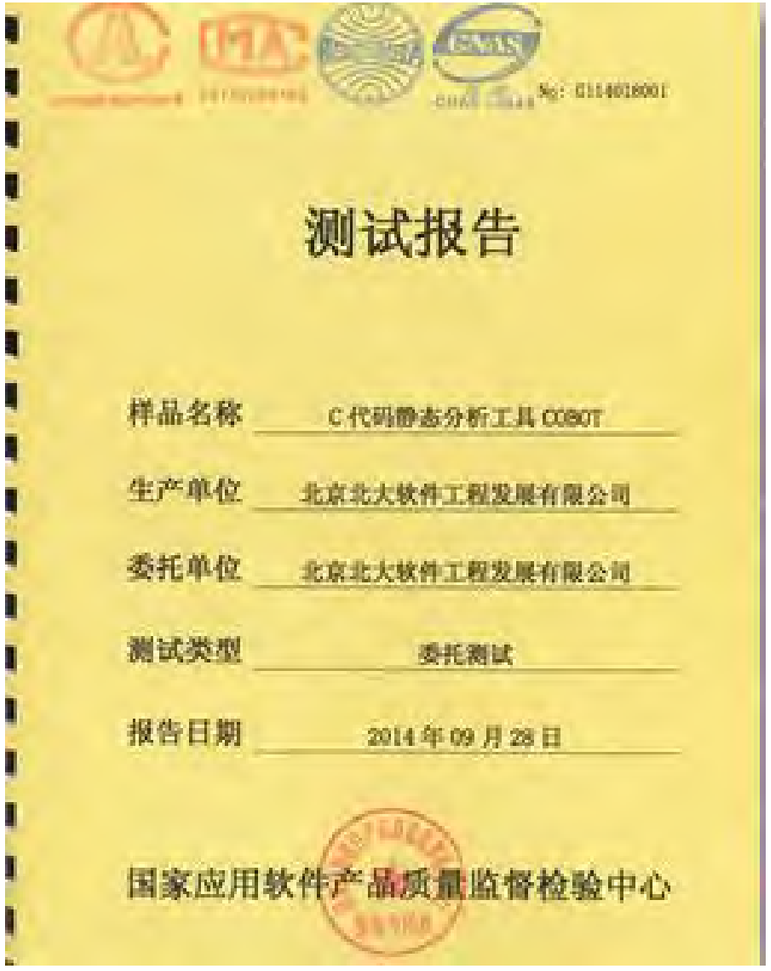
程序缺陷分析（CWE，14类110+种）

安全漏洞扫描（OWASP，8类90+种）

相关成果

- 相关技术成果发表在国际顶级会议和国内刊物上
  - ✓Safe Memory-Leak Fixing for C Programs, ICSE’15
  - ✓Fixing Recurring Crash Bugs via Analyzing Q&A Sites, ASE’15
  - ✓Practical Null Pointer Dereference detection via graph reachability, ISSRE’15
  - ✓基于值依赖分析的空指针解引用，电子学报
- 获得软件著作权8项，申请专利2项
- 2015年，北京市/国家科技创新优秀成果奖
- 中国唯一一个获得CWE Compatible的安全产品

三方测试、对比测试报告及CWE证书



No: 0114018001

测试报告

样品名称 C代码静态分析工具 CCBOT

生产单位 北京北大软件工程发展有限公司

委托单位 北京北大软件工程发展有限公司

测试类型 委托测试

报告日期 2014 年 09 月 28 日

国家应用软件产品质量监督检验中心

robot.ssrit.cn

库博 V2.5

对比测试报告

（注：此报告测试结果主要针对库博在全路通号院 9 月 4 日部署结果，版本号为 V 2.5.1）

撰写单位：北京全路通信信号研究设计院有限公司

北京北大软件工程发展有限公司

日期：2014 年 9 月

内容提要

- 引言
- 静态程序分析技术
- 基于静态分析技术的缺陷检测
- 基于静态分析技术的缺陷修复
- 总结与展望

缺陷修复技术的研究背景

- 缺陷预防
  - ✓从上世纪60年代开始
  - ✓代表技术：类型系统
- 缺陷检测
  - ✓从上世纪60年代开始
  - ✓代表技术：软件测试、软件验证
- 缺陷定位
  - ✓从上世纪90年代开始
  - ✓代表技术：统计性调试

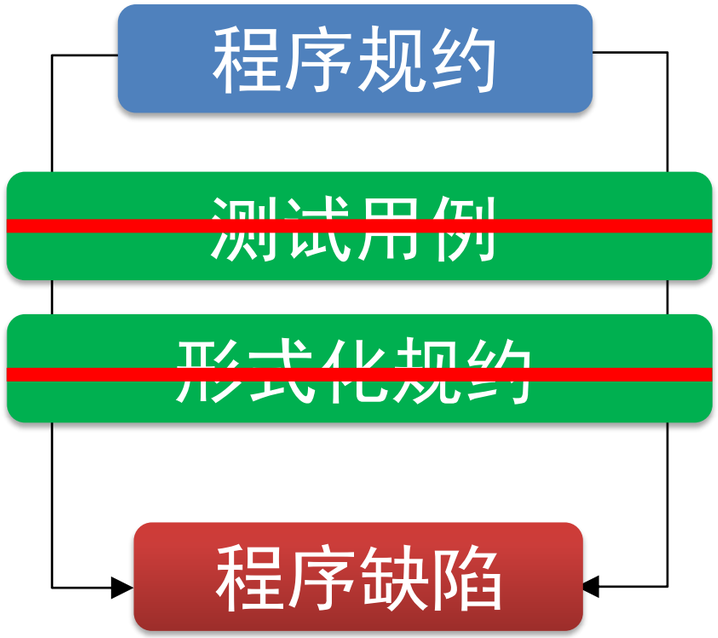
缺陷修复技术的研究背景

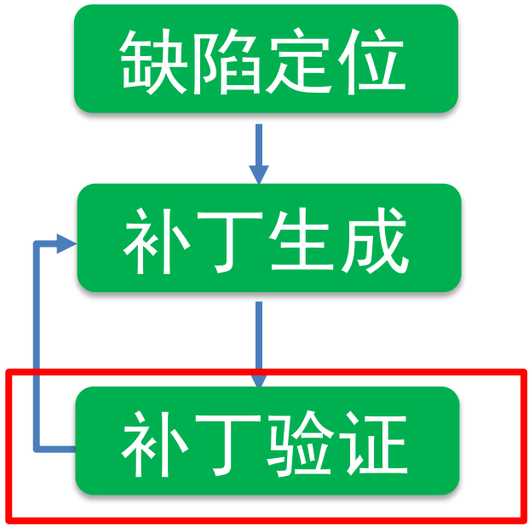
- 缺陷修复：自动消除缺陷
  - ✓从2000年开始
  - ✓代表技术：生成-验证缺陷修复技术

  - ✓2009：15篇论文
  - ✓2012：30篇论文（20+方法，3+最佳论文奖）
  - ✓数据来源：Westley Weimer: Advances in Automated Program Repair and a Call To Arms. Symposium on Search Based Software Engineering (SSBSE 2013)
  - ✓2013-2017：ICSE新增专门针对缺陷修复的session

一般类型的缺陷修复技术

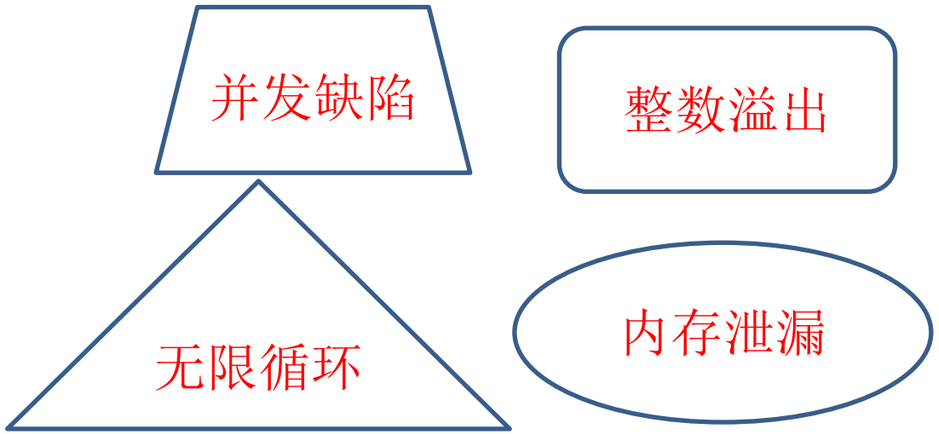
- 修复精度：30%左右





- ✓ 测试往往不充分。即便充分，用于过滤错误补丁的时间开销很大
- ✓ 形式化规约在大部分程序中不存在，且获取代价高
- ✓ 通过搜索等方式取得的补丁往往与人工写出的补丁差异较大

基于模式的自动修复技术

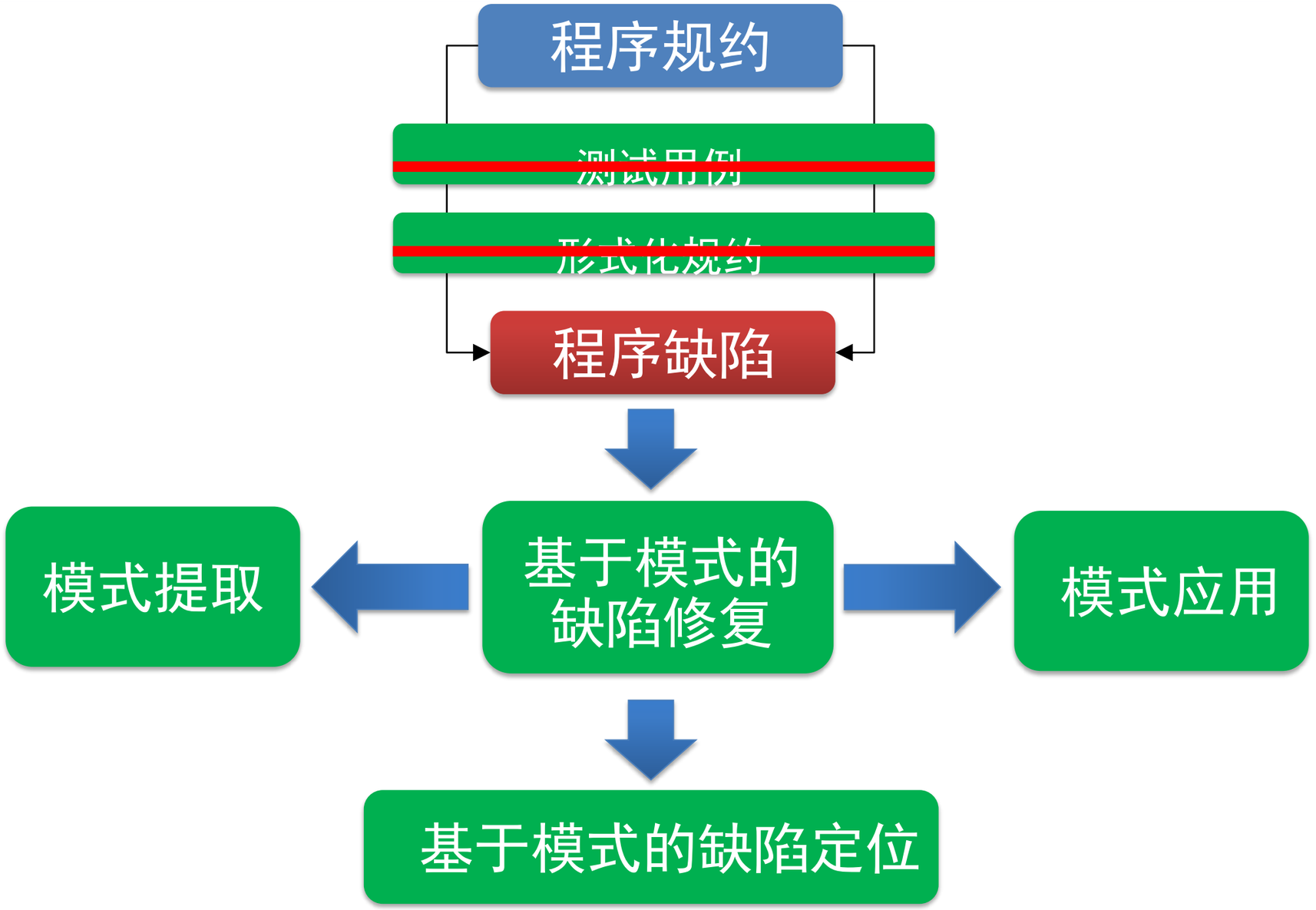


修复模式：
把已存在缺陷代码片段转换为已存在的修复后代码片段的转换操作的序列，及其对应的缺陷代码片段和修复代码片段。

```
24      @Override
25      protected void onDestroy() {
  +         super.onDestroy();
26          OpenIAB.instance().unbindService();
            //If there is something wrong with coordinates in database
27      }
```

```
127     @Override
128     protected void onDestroy() {
  +         super.onDestroy();
129         if(imgWebView != null) imgWebView.destroy();
130     }
```

基于模式的自动修复技术



基于模式的自动修复技术

- 基于模式的缺陷修复框架
  - ✓修复模式表示
  - ✓修复步骤可实例化
- 自动模式提取
  - ✓从因特网资源（问答网站）中提取模式
  - ✓基于代码分析
- 基于模式的缺陷定位
  - ✓程序崩溃缺陷：80%
  - ✓内存泄漏缺陷：100%

修复模式的表示

- 已有代码比对的表示方法
  - ✓添加
  - ✓更新
  - ✓删除
  - ✓移动
- 在修复中存在的问题
  - ✓无法描述改变的位置
  - ✓无法描述重命名的变量
- 增加的编辑操作
  - ✓替换
  - ✓拷贝

修复模式：
把已存在缺陷代码片段转换为已存在的修复后代码片段的转换操作的序列，及其对应的缺陷代码片段和修复代码片段。

修复模式的表示——改变的位置

已存在缺陷代码

```
context.registerReceiver(...);
```

1st position

修复模式：
Insert getApplicationContext()
after 1st position

已存在修复代码

```
context.getApplicationContext().registerReceiver(...);
```

缺陷代码

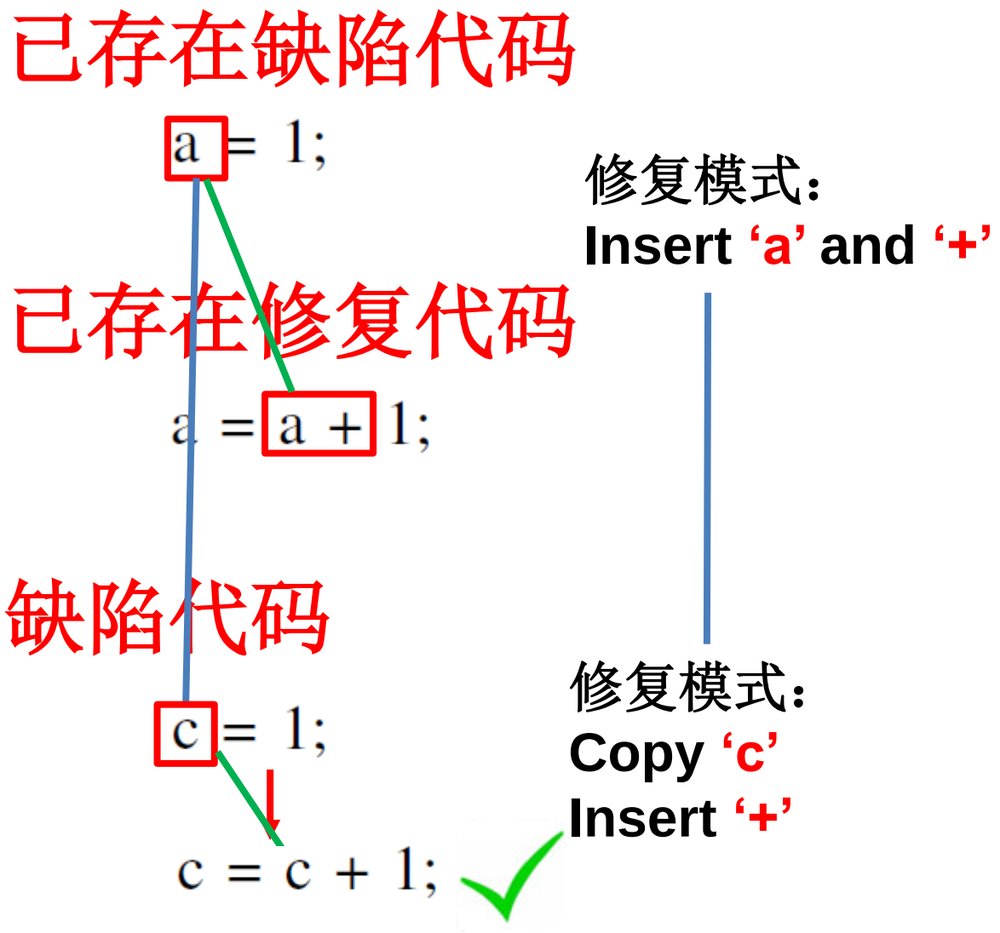
```
BatteryHelper.level(context);
```

1st position

修复模式：
Replace context with
context.getApplicationContext()

```
BatteryHelper.level(cor .getApplicationContext());
```

修复模式的表示——重命名的变量



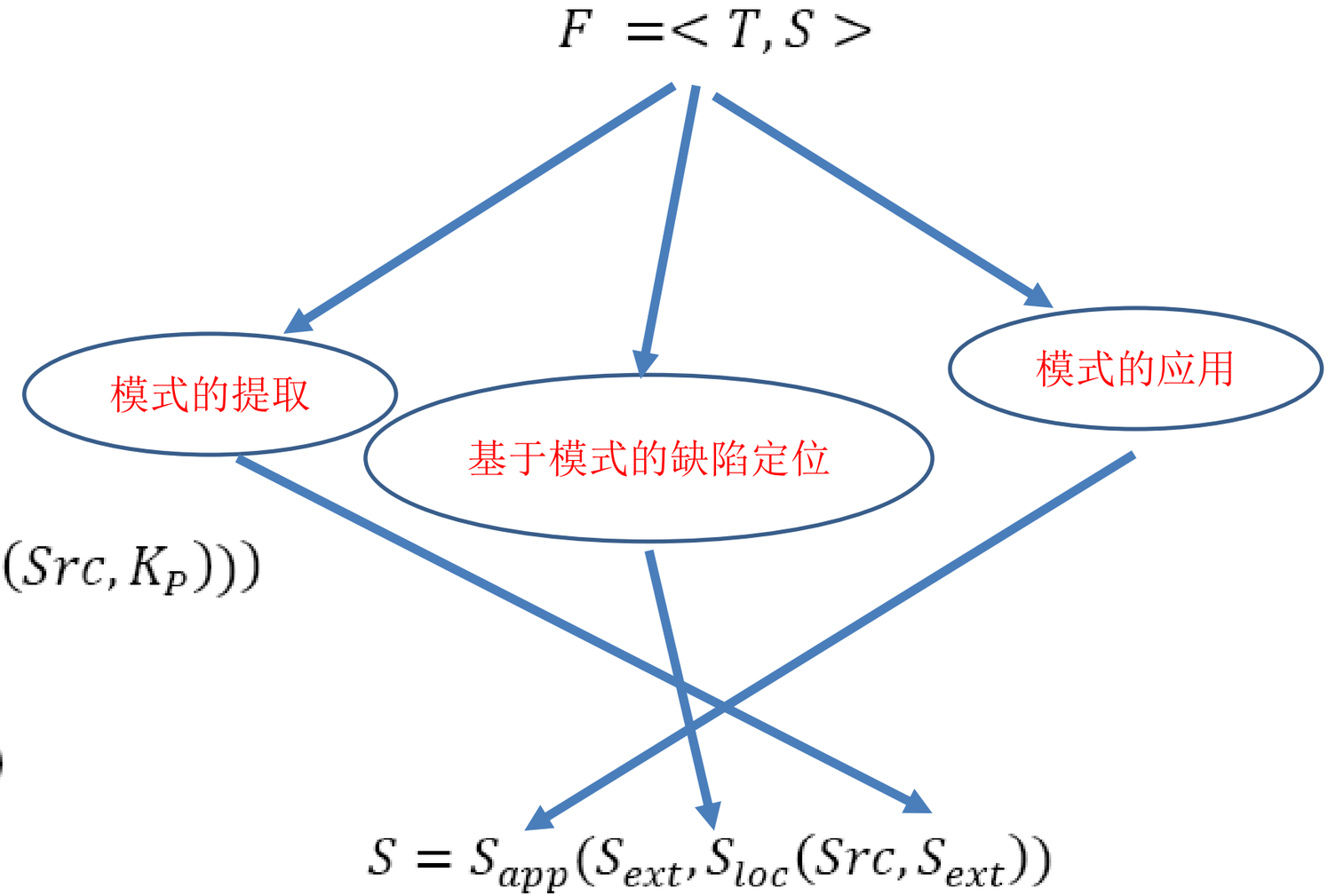
基于模式的缺陷修复框架：修复步骤

- 输入：模式来源+源代码
- 输出：补丁序列

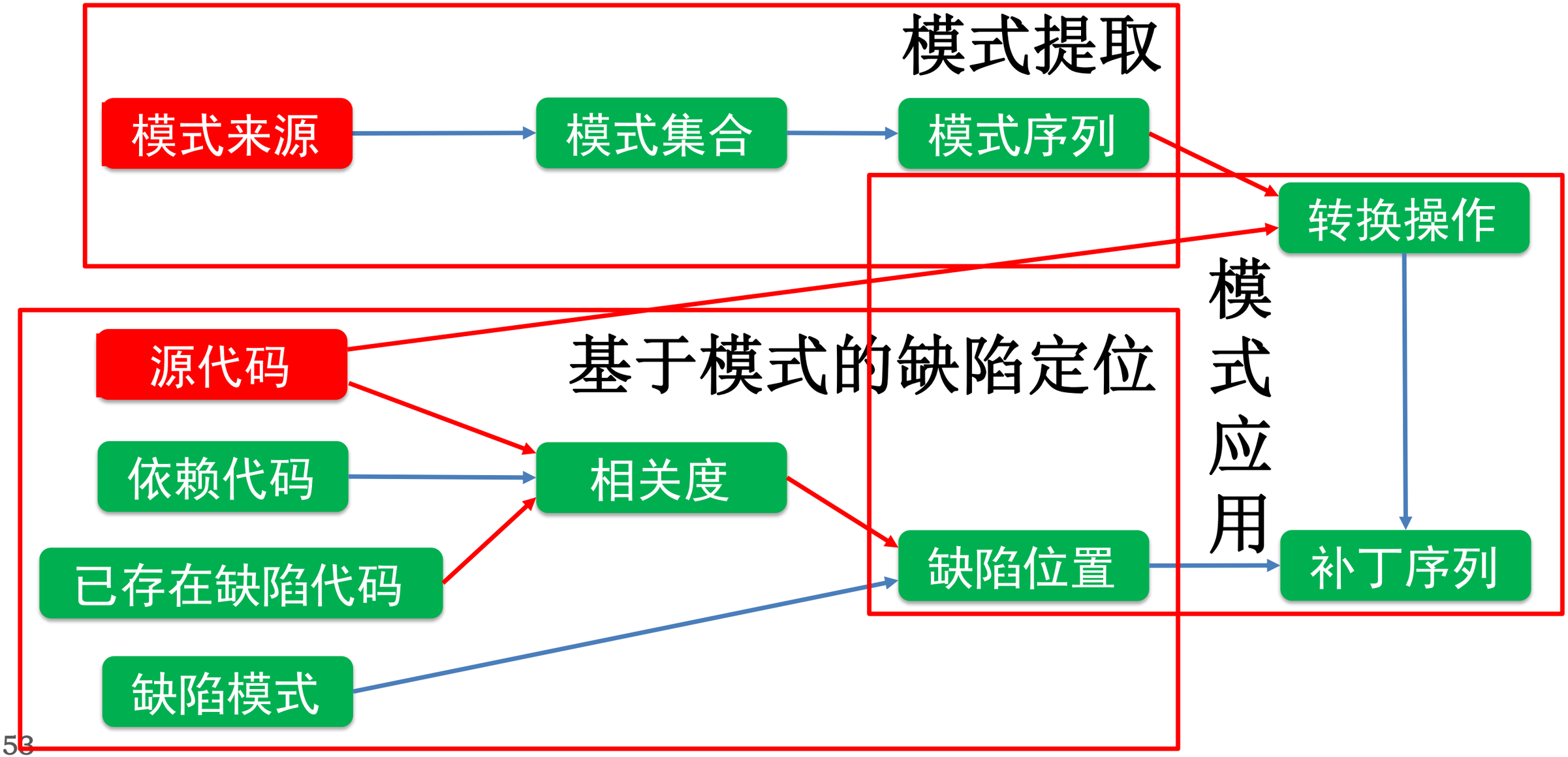
$$S_{ext} = Sort_{ext}(K_{ext} \cup A_{ext}(Orig))$$

$$S_{loc} = Sort_{loc}(K_{loc} \cup A_{loc}(K_D \cup Sim(Src, K_P)))$$

$$S_{app} = Sort_{app}(A_{app}(K_{app} \cup Map))$$



基于模式的缺陷修复框架：修复步骤



程序崩溃缺陷

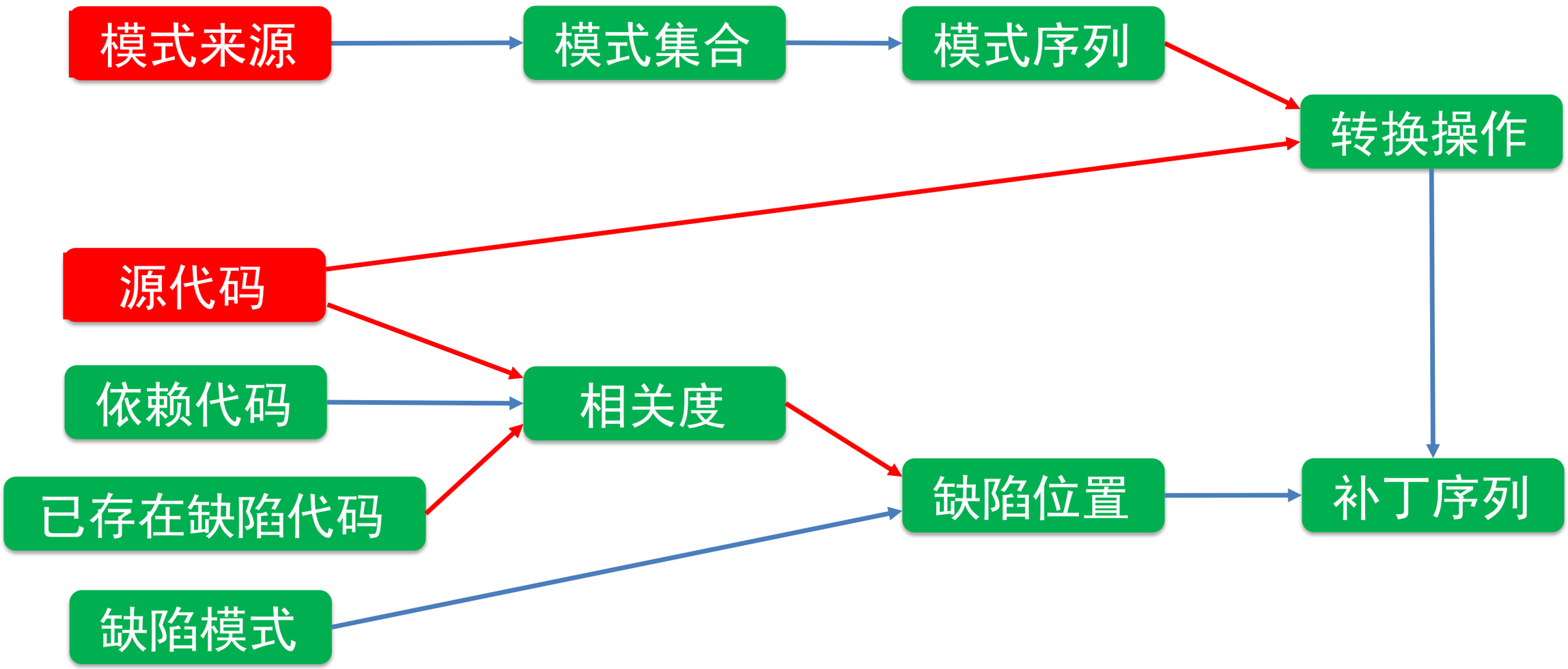
程序崩溃缺陷修复 | 内存泄漏缺陷修复

- 一类严重类型的缺陷
  - ✓ 输出停止
  - ✓ 数据丢失
- 现有工作
  - ✓ 定位
  - ✓ 预测
  - ✓ 动态修复

程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

内存泄漏缺陷修复



程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

内存泄漏缺陷修复

```
1  java.lang.RuntimeException: Unable to start receiver com.vaguehope.onosendai.update.AlarmReceiver:
   android.content.ReceiverCallNotAllowedException: IntentReceiver components are not allowed to register to receive intents
2  at android.app.ActivityThread.handleReceiver(ActivityThread.java:2126)
3  at android.app.ActivityThread.access$1500(ActivityThread.java:123)
4  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1197)
5  at android.os.Handler.dispatchMessage(Handler.java:99)
6  at android.os.Looper.loop(Looper.java:137)
7  at android.app.ActivityThread.main(ActivityThread.java:4424)
8  at java.lang.reflect.Method.invokeNative(Native Method)
9  at java.lang.reflect.Method.invoke(Method.java:511)
10 at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:784)
11 at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
12 at dalvik.system.NativeStart.main(Native Method)
13 Caused by: android.content.ReceiverCallNotAllowedException: IntentReceiver components are not allowed to register to receive intents
14 at android.app.ReceiverRestrictedContext.registerReceiver(ContextImpl.java:118)
15 at android.app.ReceiverRestrictedContext.registerReceiver(ContextImpl.java:112)
16 at com.vaguehope.onosendai.update.AlarmReceiver.onReceive(AlarmReceiver.java:31)
17 at android.app.ActivityThread.handleReceiver(ActivityThread.java:2119)
18 ... 10 more
```

已存在缺陷代码

缺陷模式

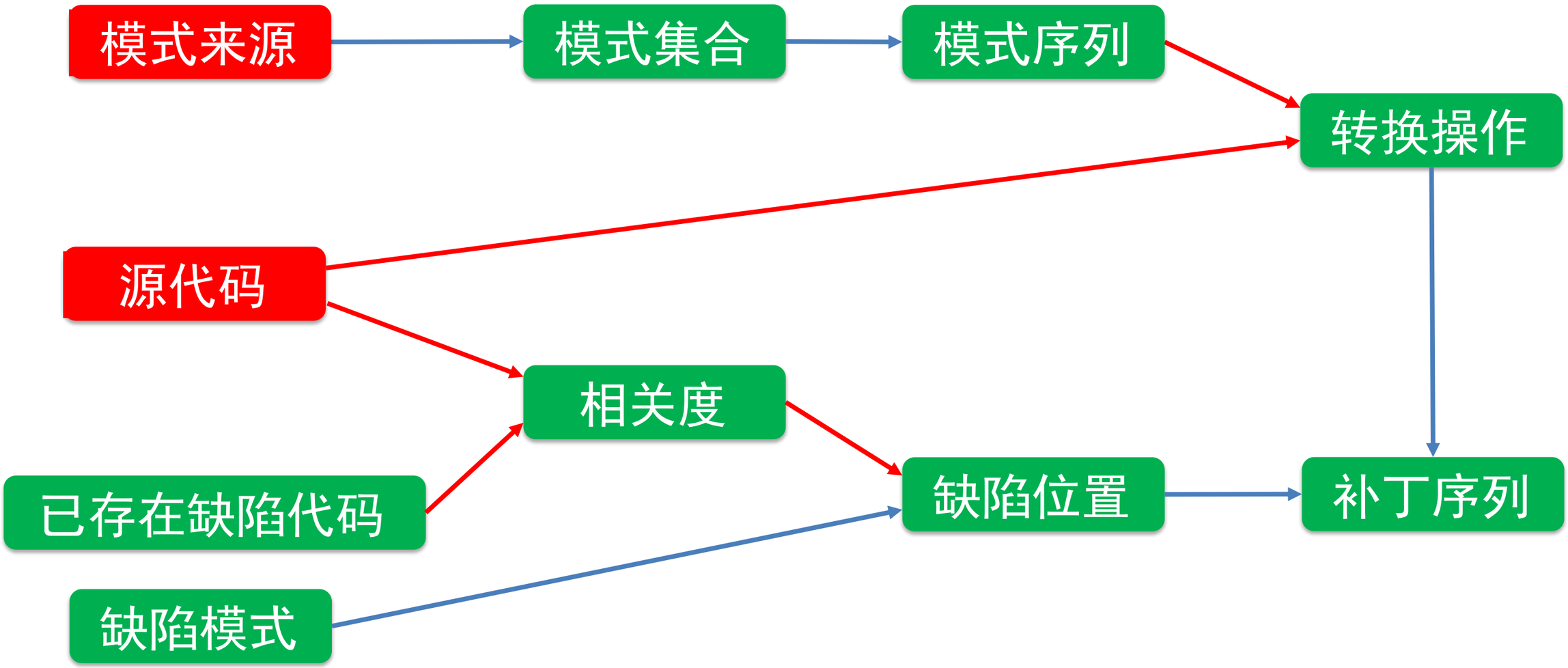
缺陷位置

补丁序列

程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

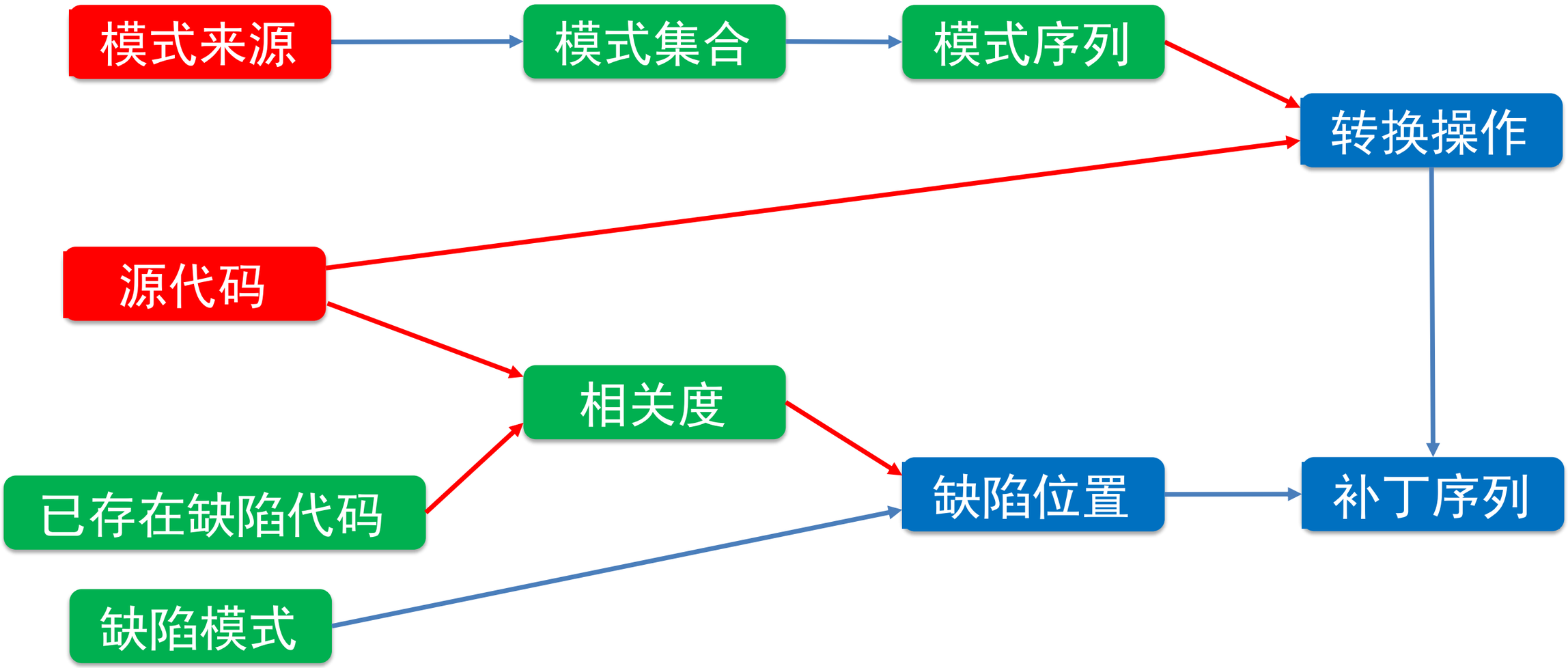
内存泄漏缺陷修复



程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

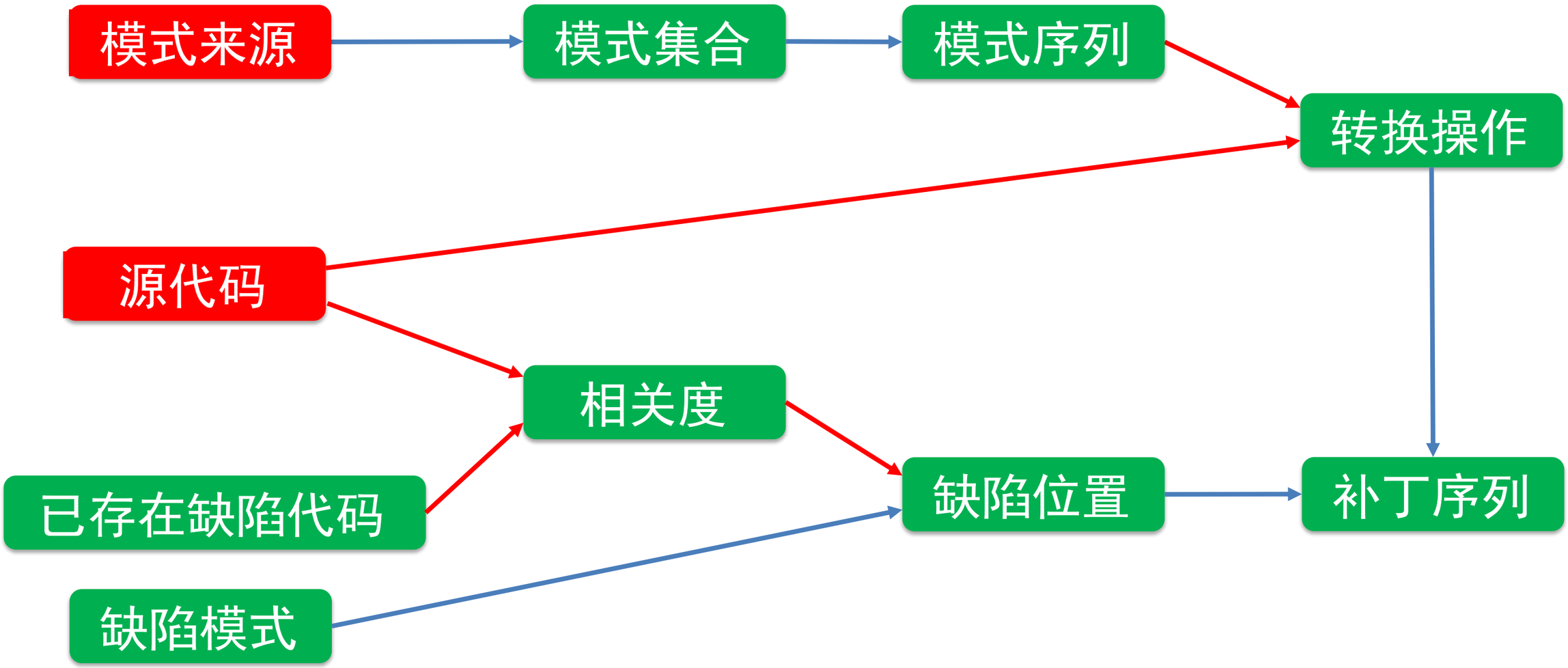
内存泄漏缺陷修复



程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

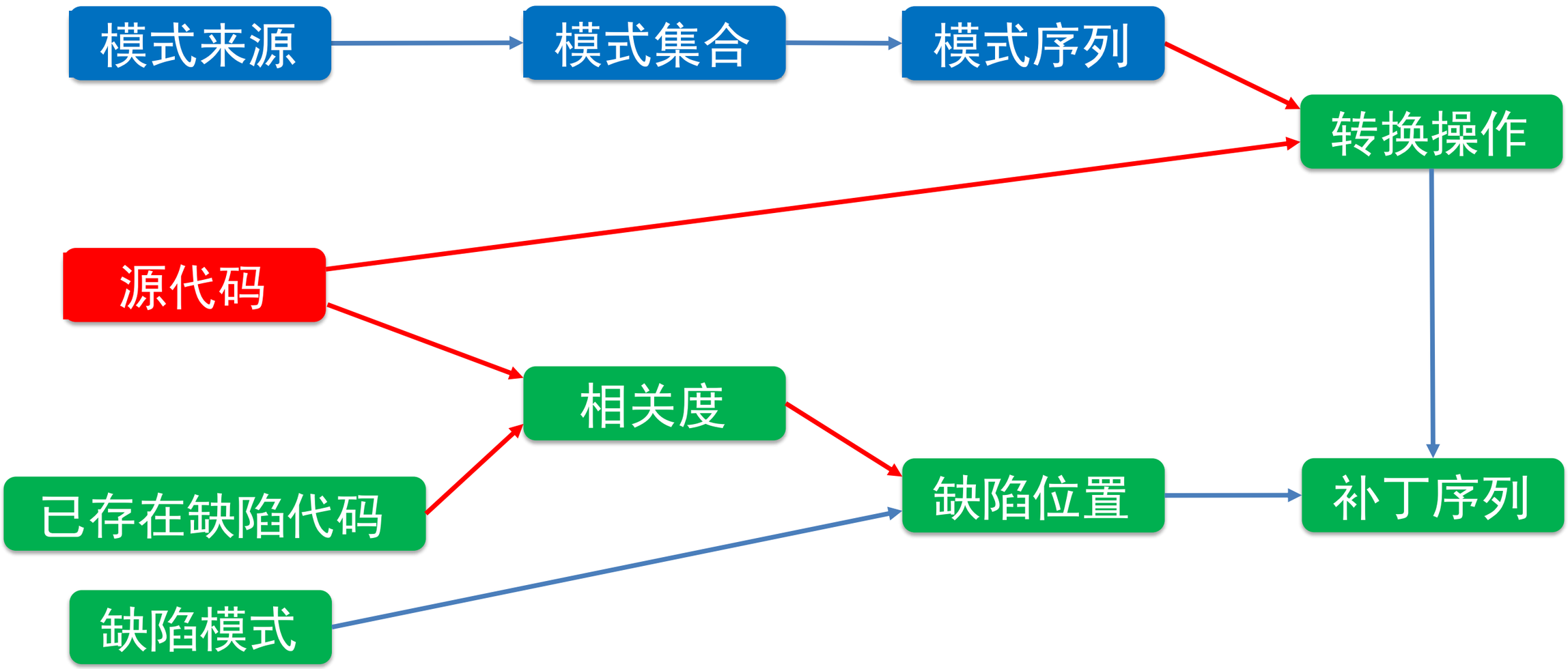
内存泄漏缺陷修复



程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

内存泄漏缺陷修复



自动模式提取

程序崩溃缺陷修复

内存泄漏缺陷修复

- 思想
  - ✓分析问答网站（Stack Overflow）
  - ✓分析代码而不使用复杂的自然语言处理技术

自动模式提取

程序崩溃缺陷修复

内存泄漏缺陷修复

```
29    public void onReceive (final Context context, final Intent intent) {
30        final int action = intent.getExtras().getInt(KEY_ACTION, -1);
31        final float bl = BatteryHelper.level(context);
32        LOG.i("AlarmReceiver invoked: action=%s bl=%s.", action, bl);
33        switch (action) {
...           ...
51        }
```

```
1  java.lang.RuntimeException: Unable to start receiver com.vaguehope.onosendai.update.AlarmReceiver:
   android.content.ReceiverCallNotAllowedException: IntentReceiver components are not allowed to register to receive intents
2  at android.app.ActivityThread.handleReceiver(ActivityThread.java:2126)
3  at android.app.ActivityThread.access$1500(ActivityThread.java:123)
4  at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1197)
5  at android.os.Handler.dispatchMessage(Handler.java:99)
6  at android.os.Looper.loop(Looper.java:137)
7  at android.app.ActivityThread.main(ActivityThread.java:4424)
8  at java.lang.reflect.Method.invokeNative(Native Method)
9  at java.lang.reflect.Method.invoke(Method.java:511)
10 at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:784)
11 at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
12 at dalvik.system.NativeStart.main(Native Method)
13 Caused by: android.content.ReceiverCallNotAllowedException: IntentReceiver components are not allowed to register to receive intents
14 at android.app.ReceiverRestrictedContext.registerReceiver(ContextImpl.java:118)
15 at android.app.ReceiverRestrictedContext.registerReceiver(ContextImpl.java:112)
16 at com.vaguehope.onosendai.update.AlarmReceiver.onReceive(AlarmReceiver.java:31)
17 at android.app.ActivityThread.handleReceiver(ActivityThread.java:2119)
18 ... 10 more
```

context: C

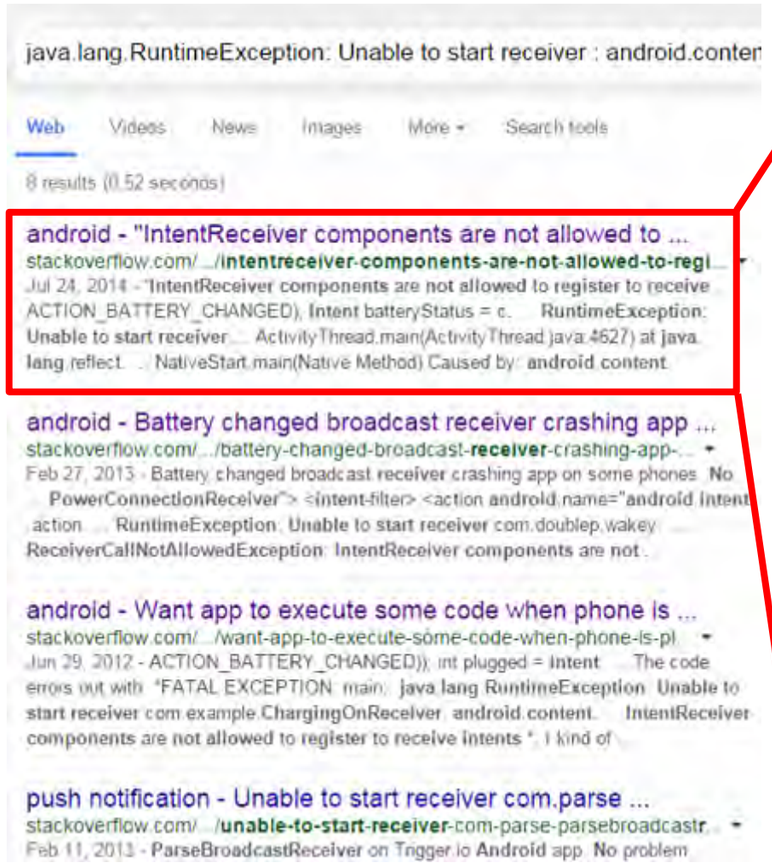
应该为

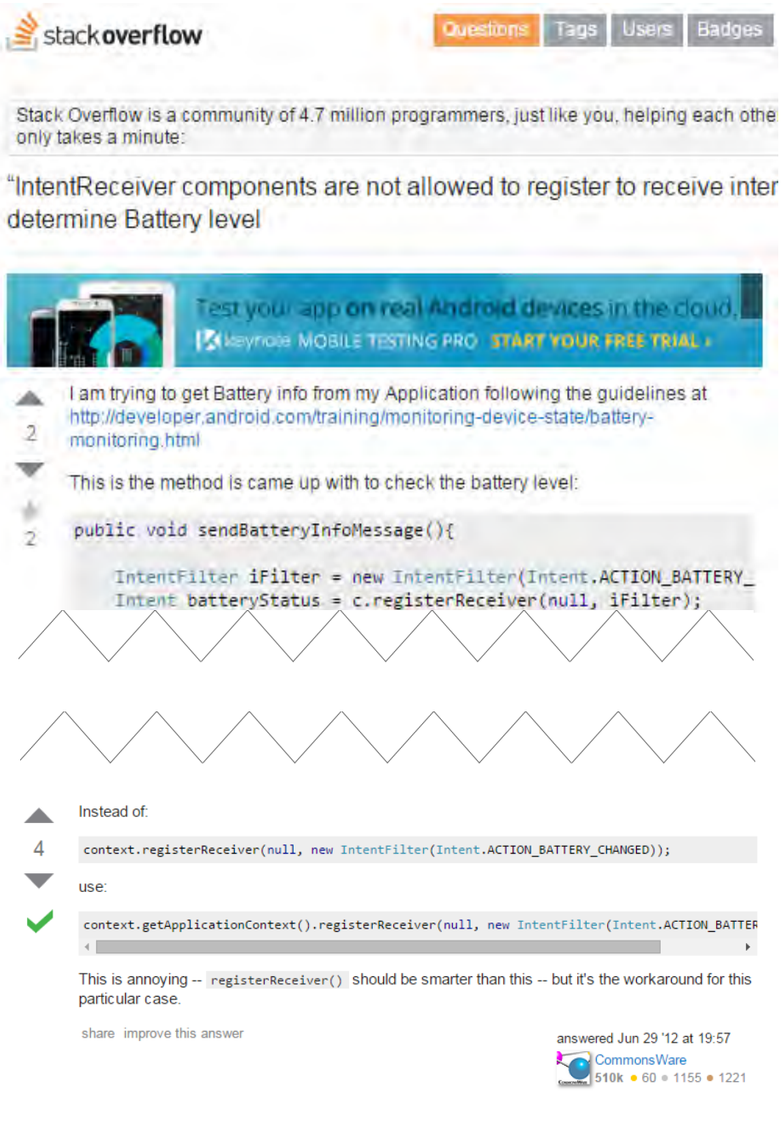
context.ge

自动模式提取

程序崩溃缺陷修复

内存泄漏缺陷修复





自动模式提取

程序崩溃缺陷修复

内存泄漏缺陷修复

```
29    public void onReceive (final Context context, final Intent intent) {
30        final int action = intent.getExtras().getInt(KEY_ACTION, -1);
31        final float bl = BatteryHelper.level(context);
32        LOG.i("AlarmReceiver invoked: action=%s bl=%s.", action, bl);
33        switch (action) {
...           ...
51        }
52    }
```

```
final float bl=BatteryHelper.level(
                context.getApplicationContext());
```

✓ 修复

自动模式提取

程序崩溃缺陷修复

内存泄漏缺陷修复

Instead of:

4

```
context.registerReceiver(null, new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
```

use:

```
context.getApplicationContext().registerReceiver(null, new IntentFilter(Intent.ACTION_BATTER
```

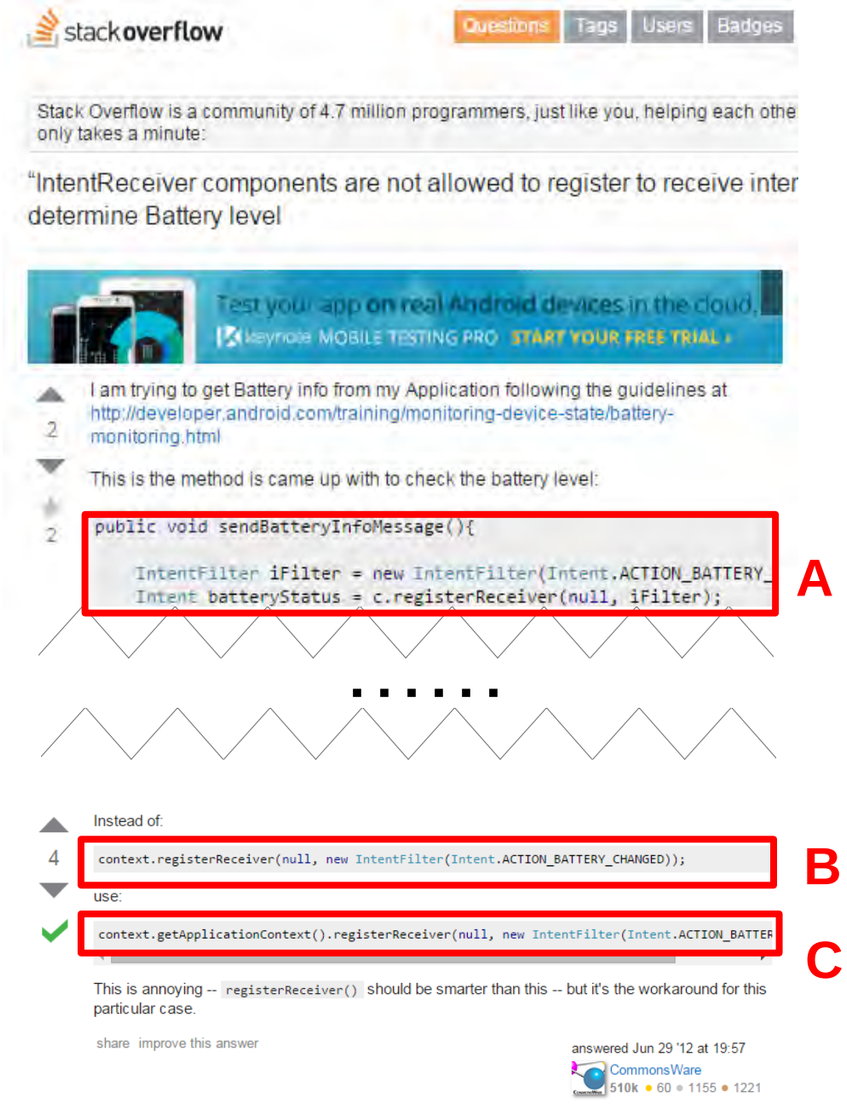
This is annoying -- `registerReceiver()` should be smarter than this -- but it's the workaround for this particular case.

- A|B
- A|C

自动模式提取

程序崩溃缺陷修复

内存泄漏缺陷修复



- B|C
- A|B
- A|C

实验

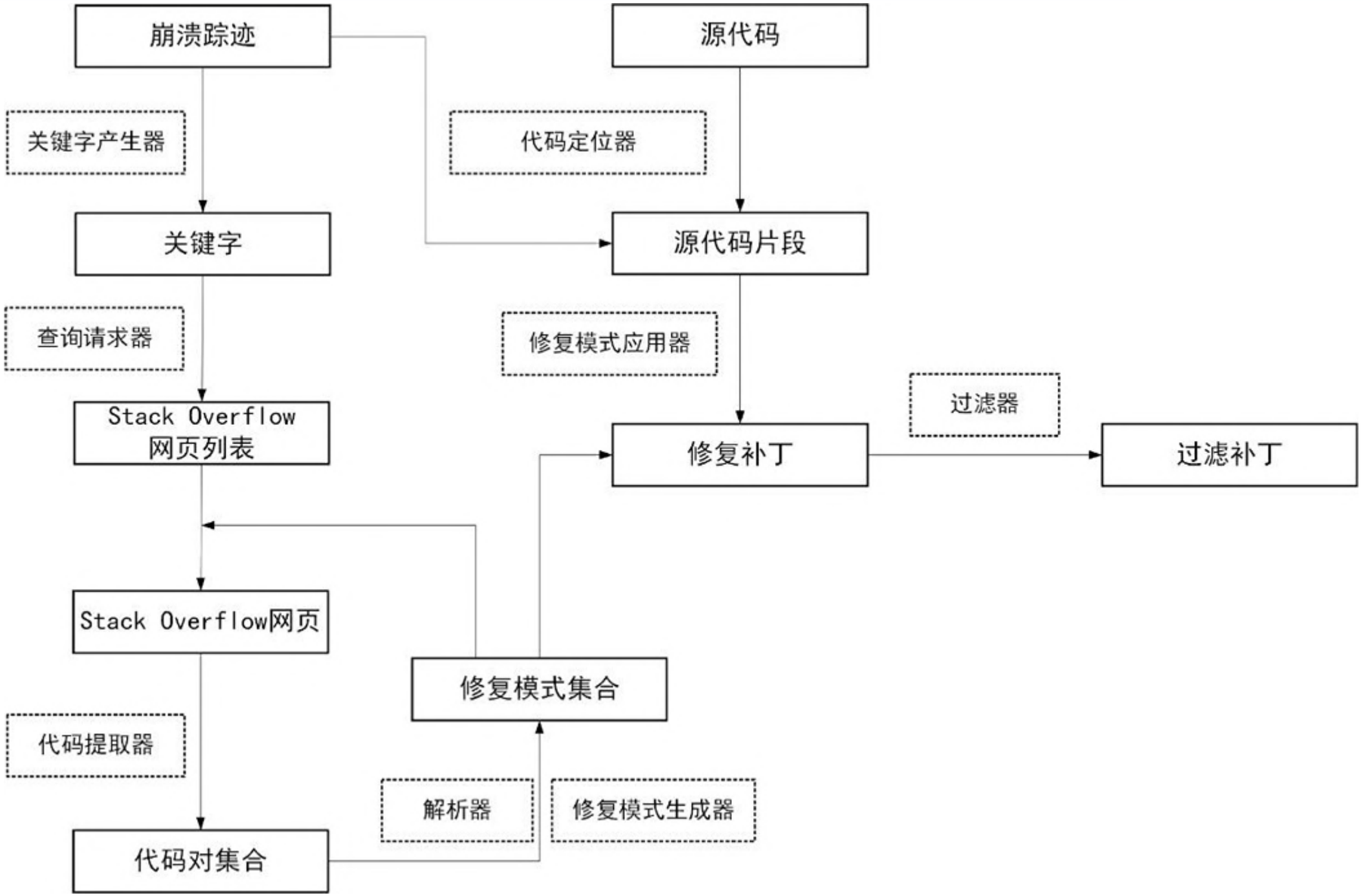
程序崩溃缺陷修复 | 内存泄漏缺陷修复

- RQ1:有效性
  - ✓我们的方法在修复真实世界的复发缺陷上有效性有多高?
- RQ2:有用性
  - ✓我们的方法是否能补充现有的修复方法?
- 实验对象: GitHub项目
- 复发缺陷: 24
  - ✓Android工程
  - ✓崩溃踪迹
  - ✓人工写出的补丁
  - ✓解决方案在因特网中已经存在

工具设计

程序崩溃缺陷修复

内存泄漏缺陷修复



实验结果

程序崩溃缺陷修复　内存泄漏缺陷修复

- RQ1:有效性
  - ✓24个缺陷中产生了10个(第一个)补丁，其中8个正确
  - ✓37.5 s/bug

- RQ2:有用性

Issue	Grep Command	Result
TuCanMobile #27	grep ”isShowing” -R .	N
OpenIAB #62	grep ”super.onDestroy” -R .	N
Onosendai #100	grep ”context.getApplicationContext” -R .	N
open-keychain #217	grep ”dismissAllowing” -R .	N
cgeo #887	grep ”image/jpeg” -R .	N
cgeo #887	grep ”image/*” -R .	N
LNReader-Android #62	grep ”super.onDestroy” -R .	N
Wordpress-Android #1320	grep ”commitAllowingStateLoss” -R .	N
cgeo #3991	grep ”isFinishing” -R .	N
cgeo #3991	grep ”\btry\b” -R .	Y
cgeo #3991	grep ”\bcatch\b” -R .	Y

修复示例

程序崩溃缺陷修复

内存泄漏缺陷修复

```
24      @Override
25      protected void onDestroy() {
  +         super.onDestroy();
26          OpenIAB.instance().unbindService();
            //If there is something wrong with coordinates in database
27      }
```

```
24      @Override
25      protected void onDestroy() {
  +         super.onDestroy();
26          OpenIAB.instance().unbindService();
            //If there is something wrong with coordinates in database
27      }
```

```
637     ft.replace(R.id.layout_fragment_container, readerFragment, tagForFragment)
638         .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE)
639         .addToBackStack(tagForFragment)
640 -       .commit();
    +       .commitAllowingStateLoss();
```

```
637     ft.replace(R.id.layout_fragment_container, readerFragment, tagForFragment)
638         .setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE)
639         .addToBackStack(tagForFragment)
640 -       .commit();
    +       .commitAllowingStateLoss();
```

内存泄漏缺陷

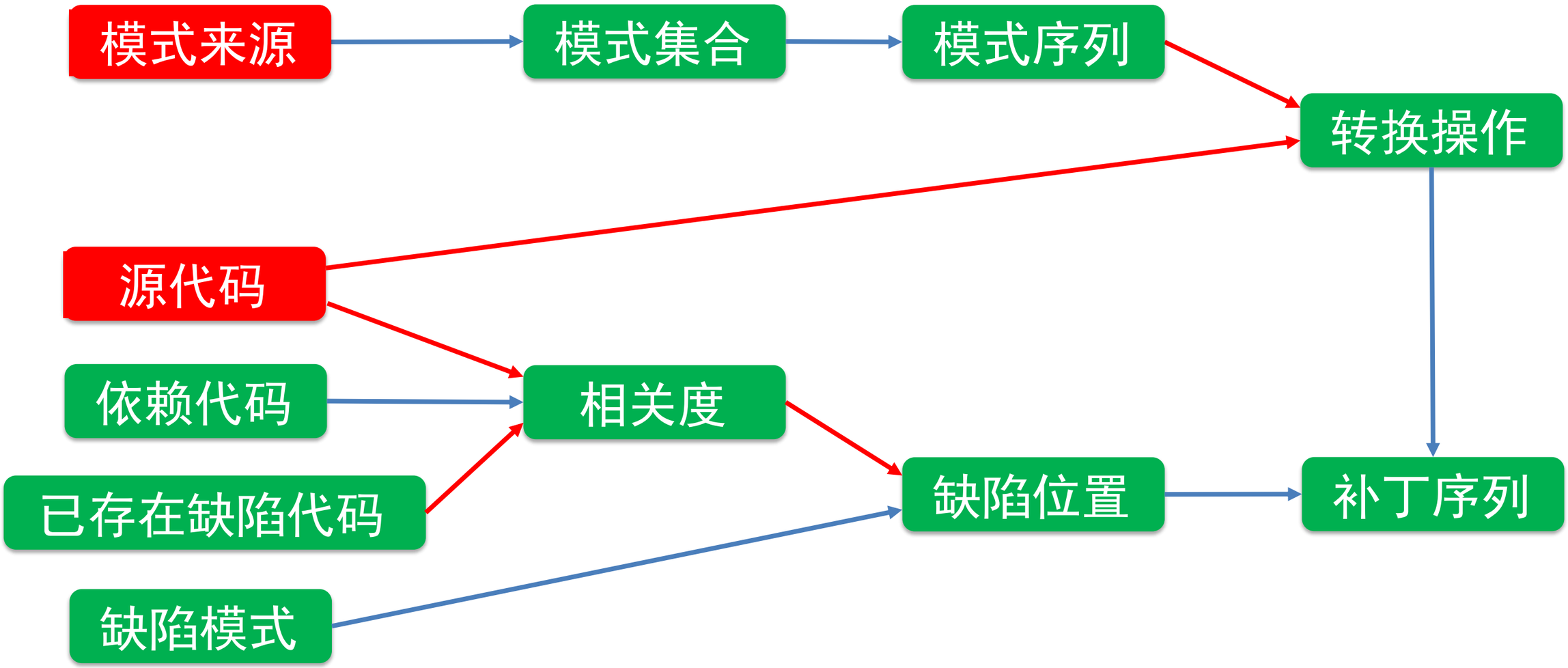
程序崩溃缺陷修复 | 内存泄漏缺陷修复

- 一类严重类型的缺陷
  - ✓影响多个应用程序
  - ✓广泛存在
- 现有研究只关注检测，没有修复
- C语言没有垃圾回收机制

内存泄漏缺陷的修复步骤

程序崩溃缺陷修复

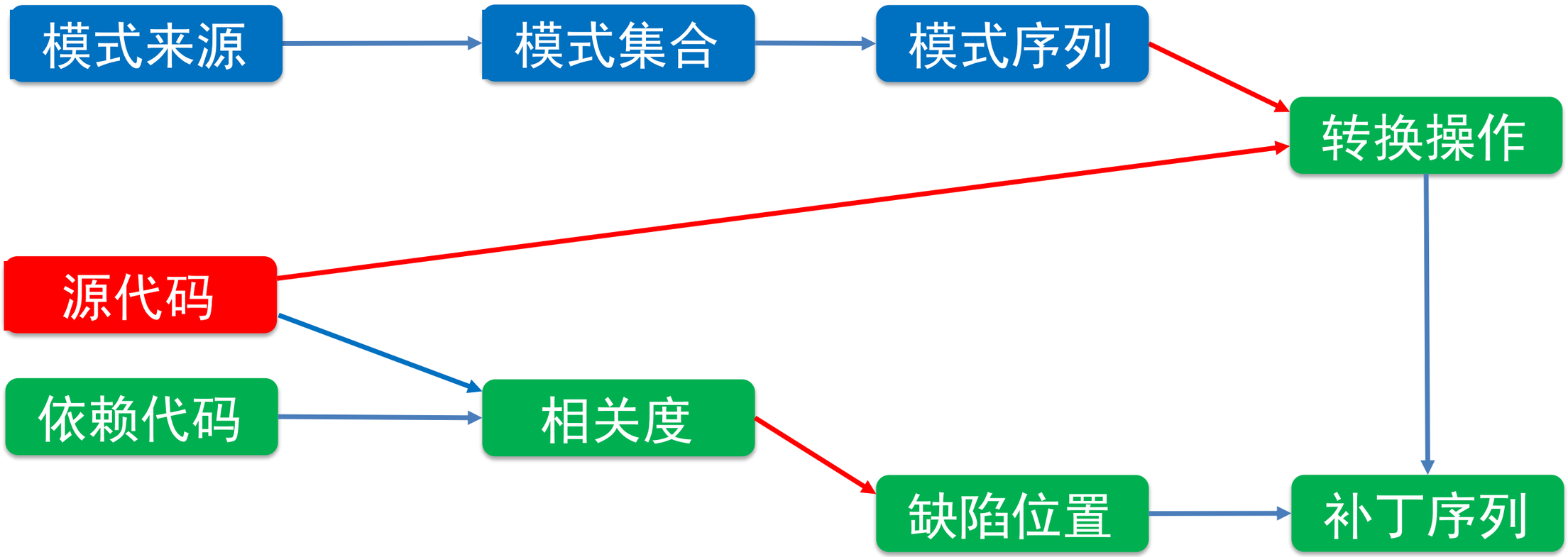
内存泄漏缺陷修复



内存泄漏缺陷的修复步骤

程序崩溃缺陷修复

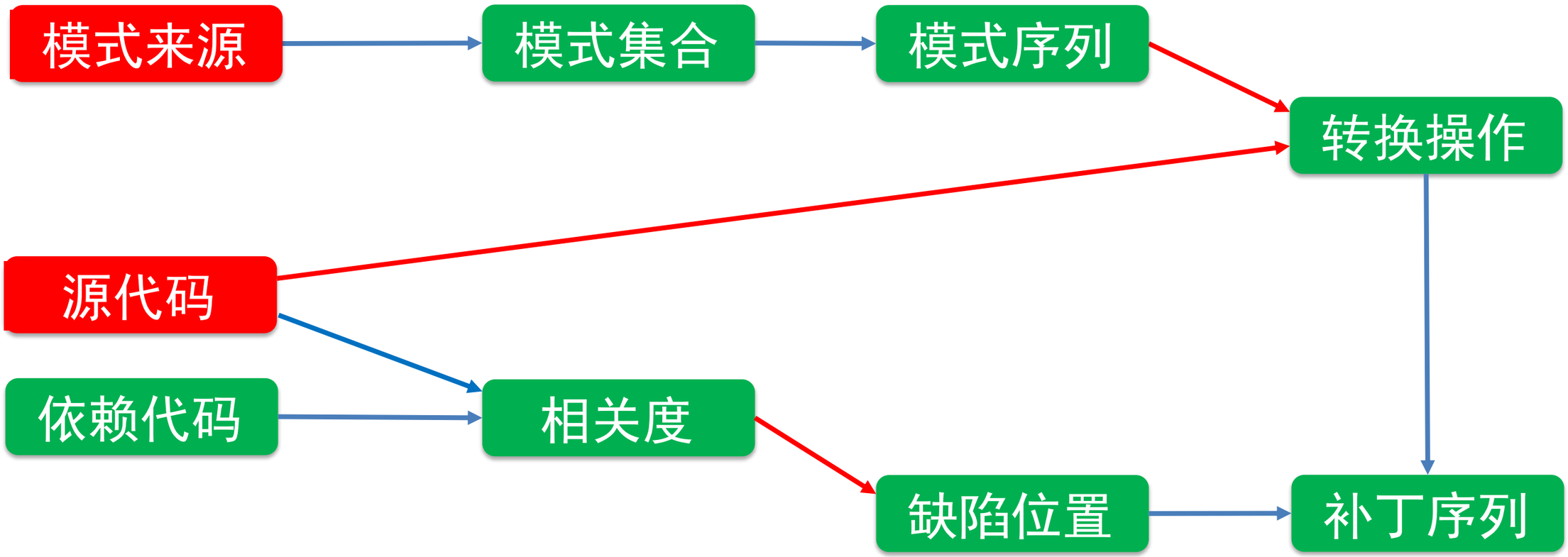
内存泄漏缺陷修复



内存泄漏缺陷的修复步骤

程序崩溃缺陷修复

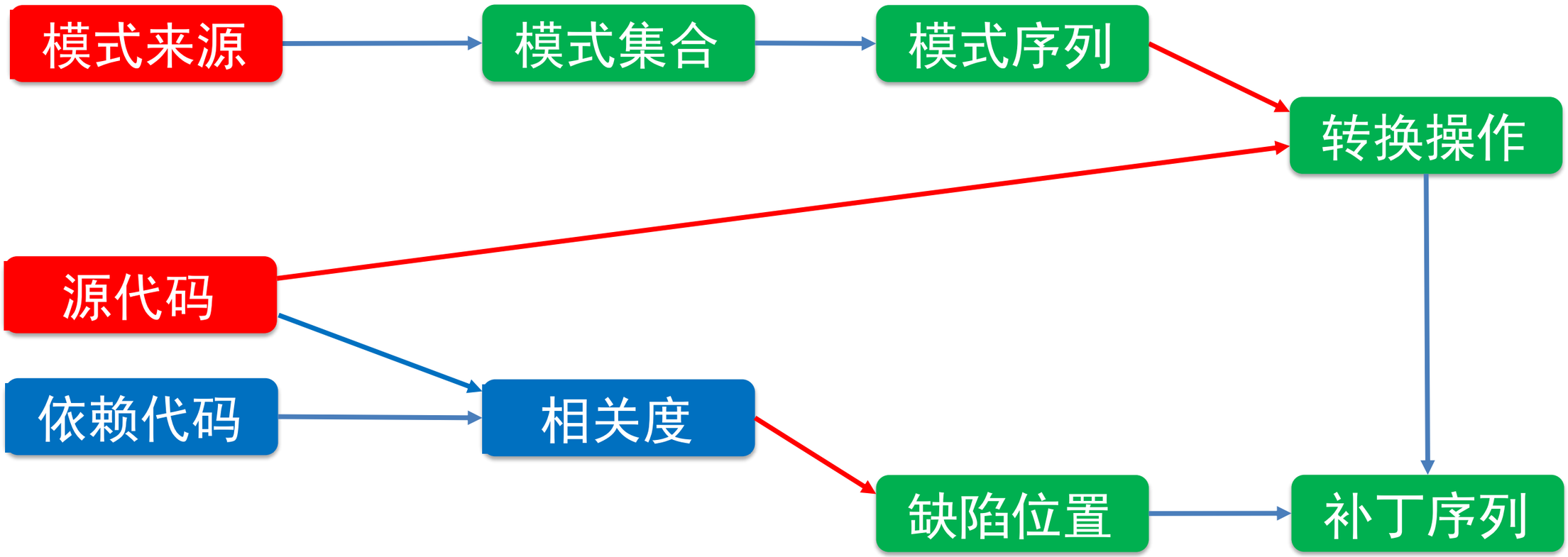
内存泄漏缺陷修复



内存泄漏缺陷的修复步骤

程序崩溃缺陷修复

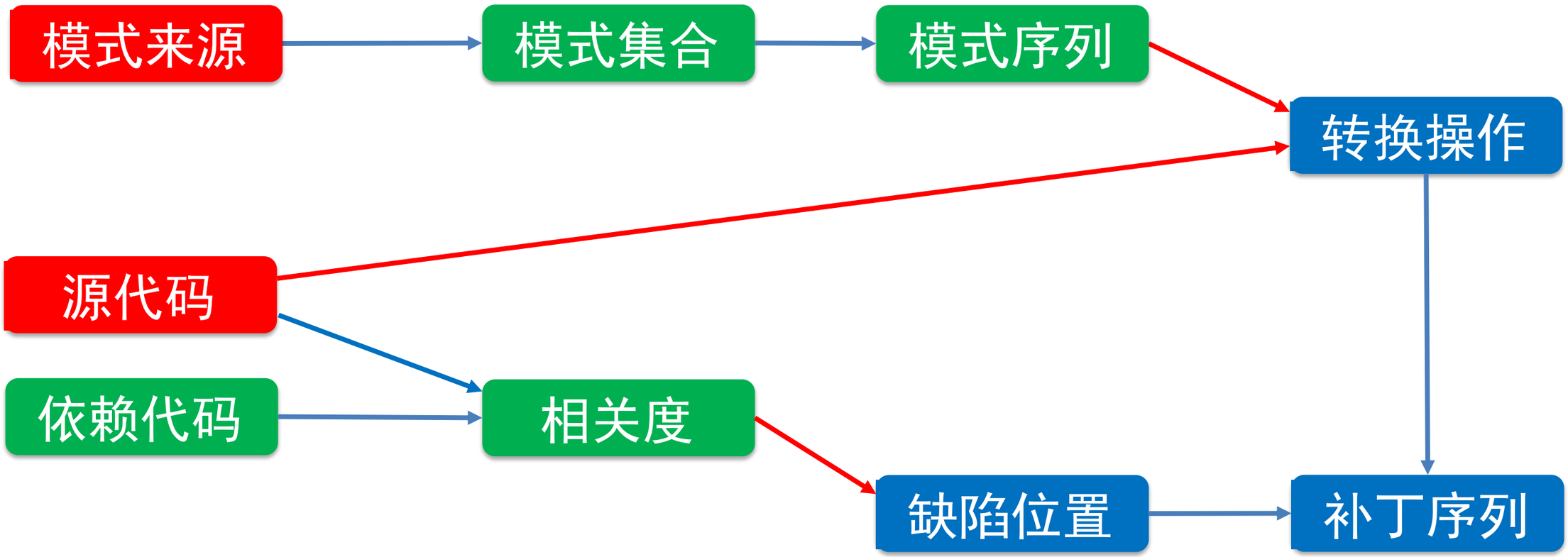
内存泄漏缺陷修复



内存泄漏缺陷的修复步骤

程序崩溃缺陷修复

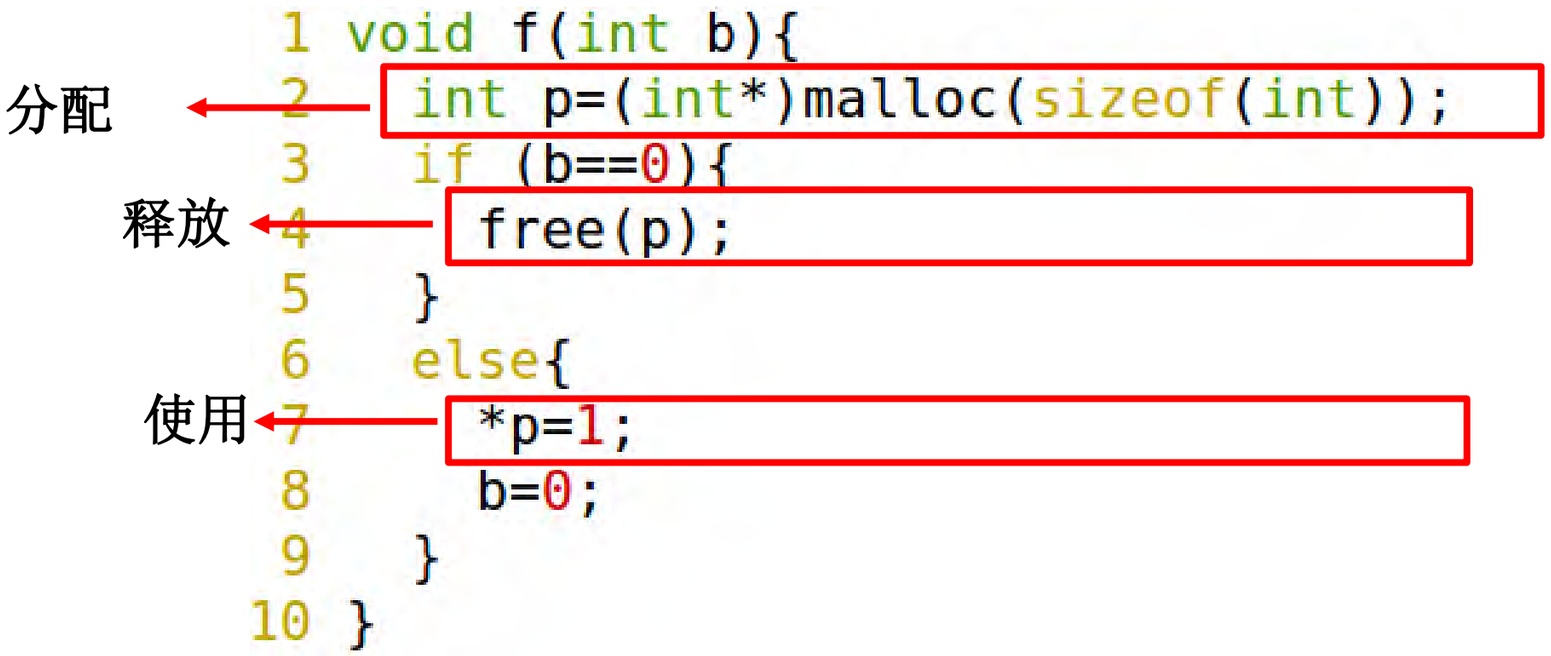
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

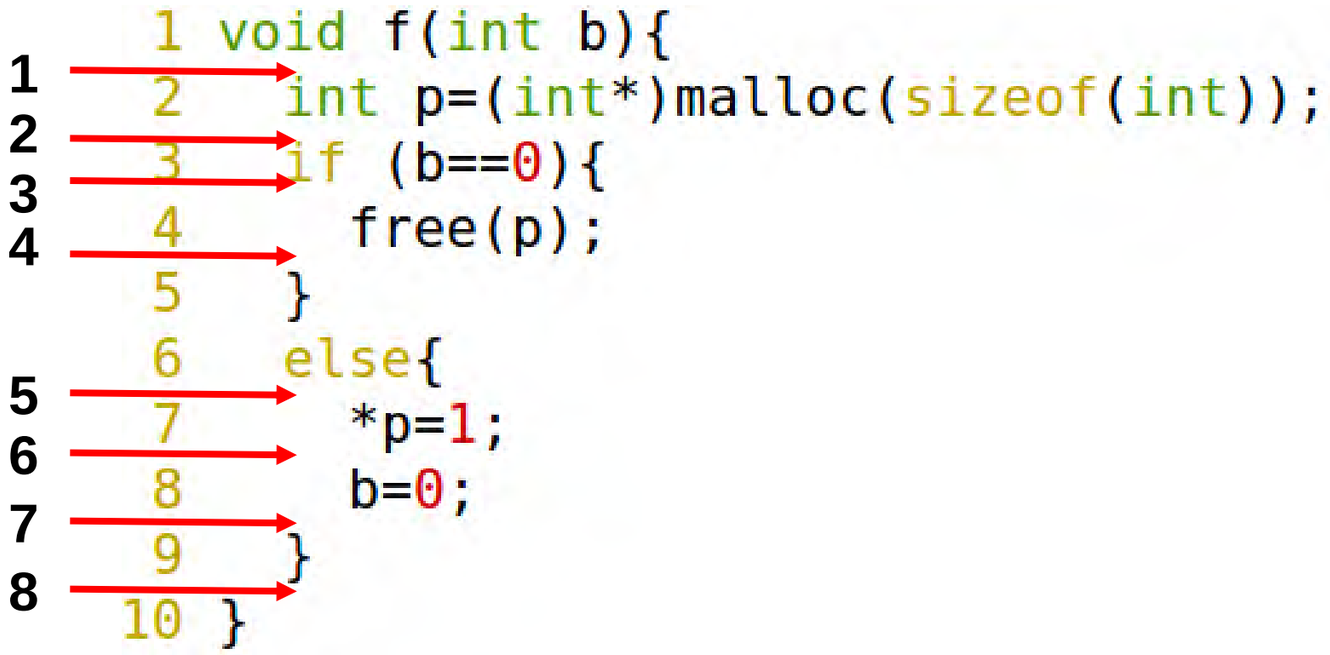
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

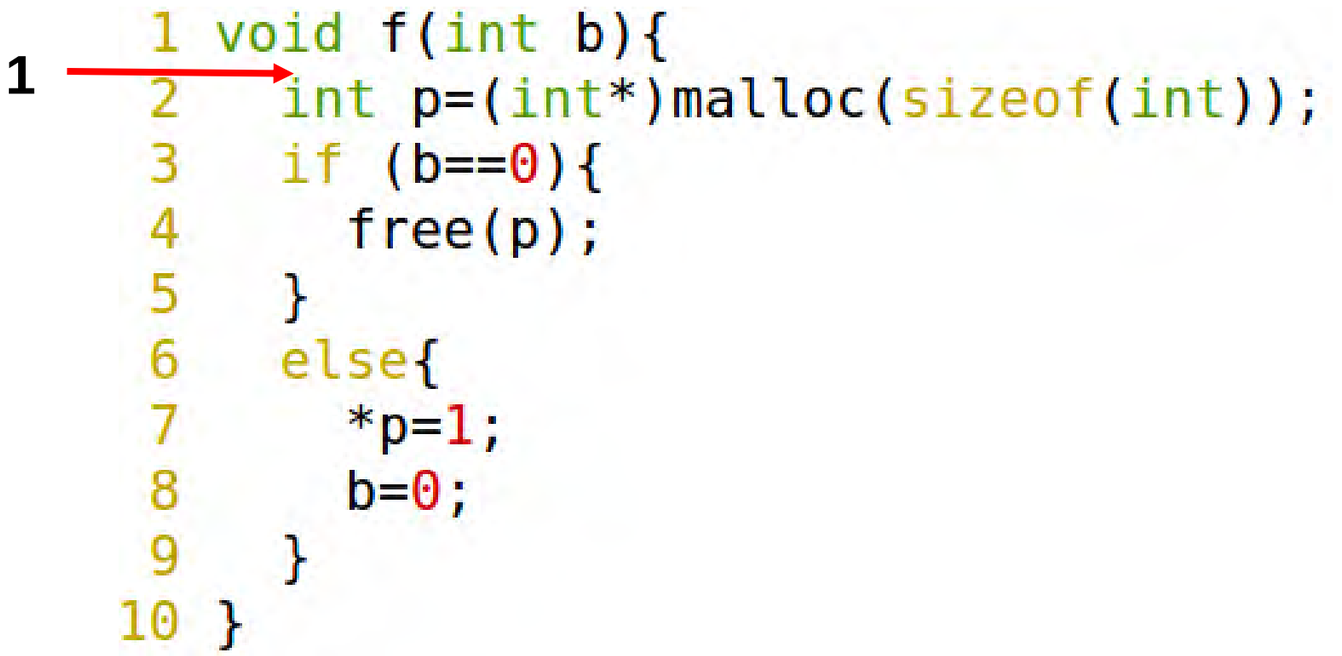
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

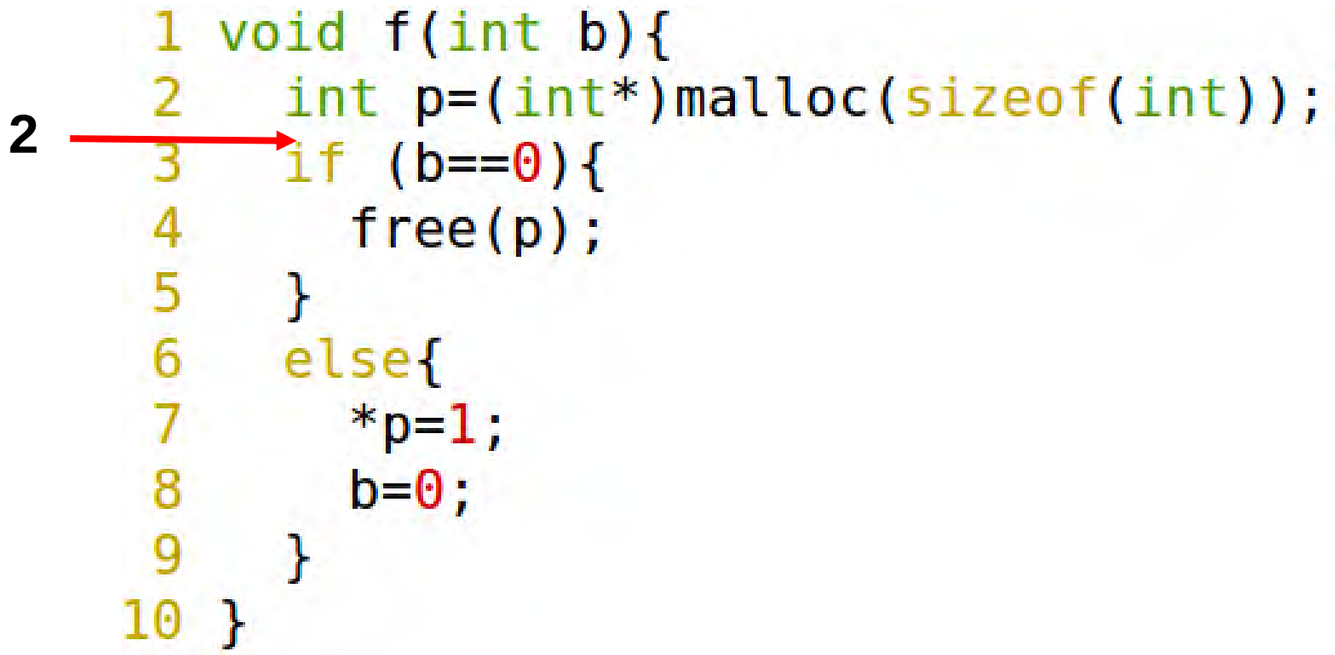
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

内存泄漏缺陷修复

```
1  void f(int b){
2    int p=(int*)malloc(sizeof(int));
3    if (b==0){
4      free(p);
5    }
6    else{
7      *p=1;
8      b=0;
9    }
10 }
```

3

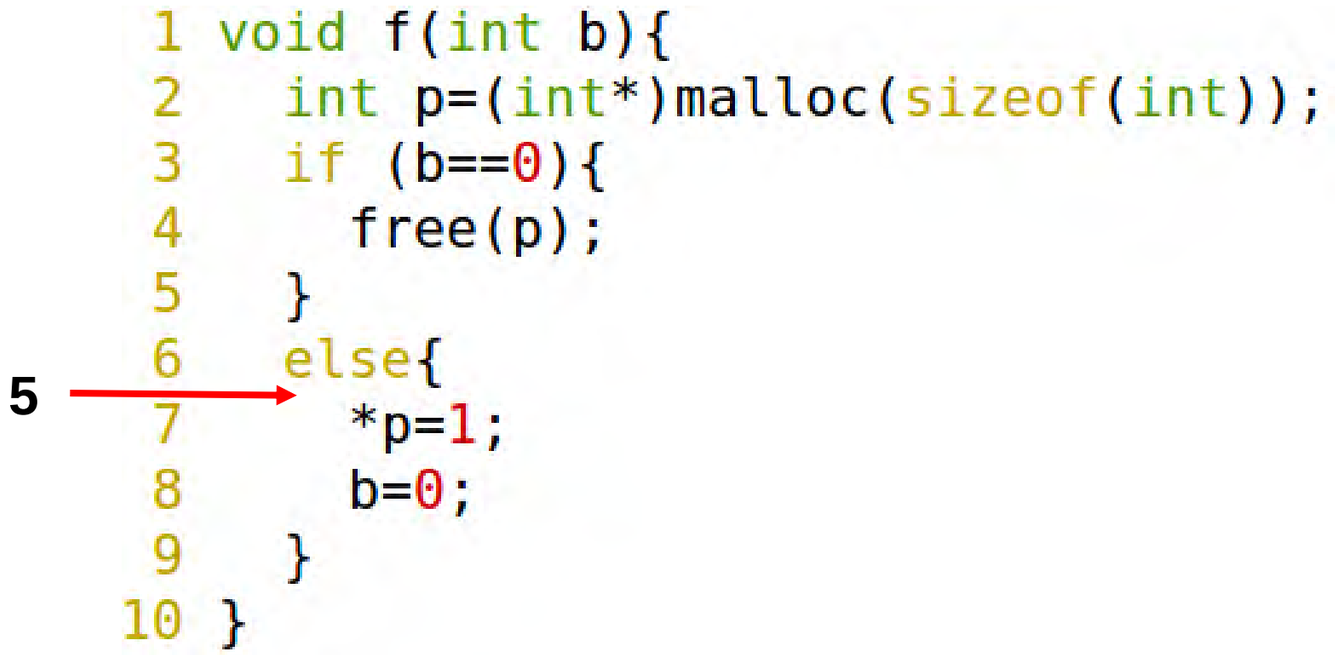
4

8

问题的提出——示例

程序崩溃缺陷修复

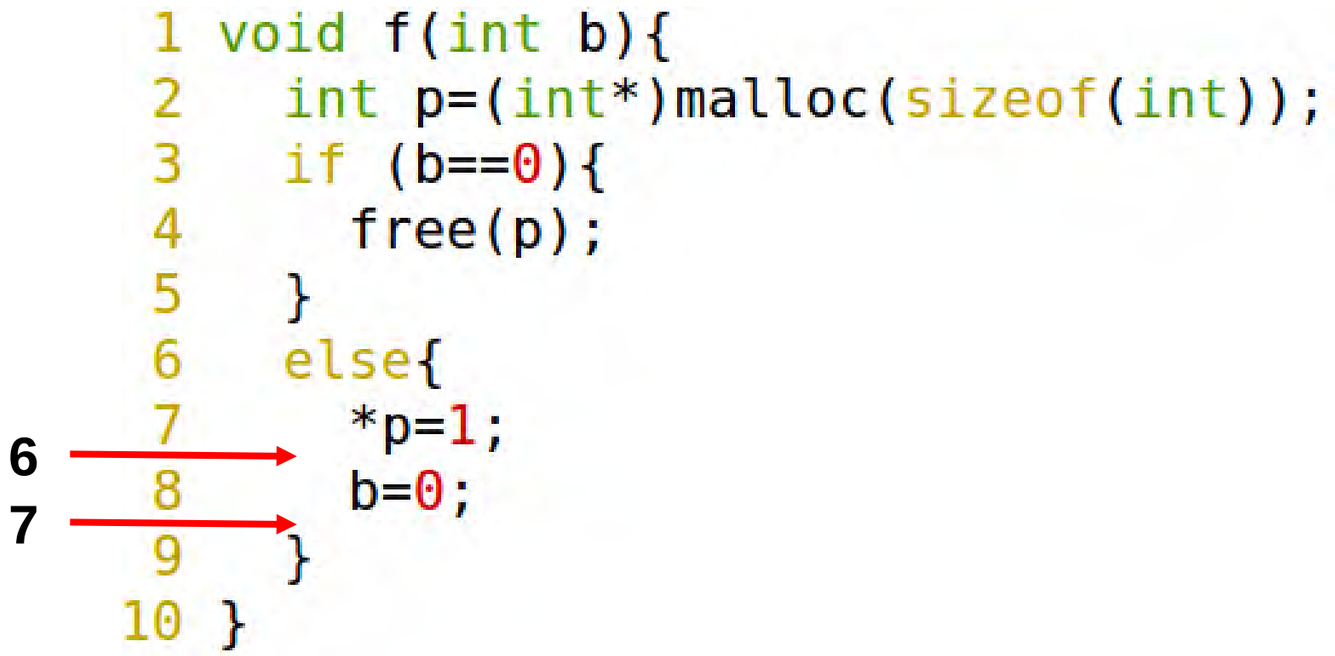
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

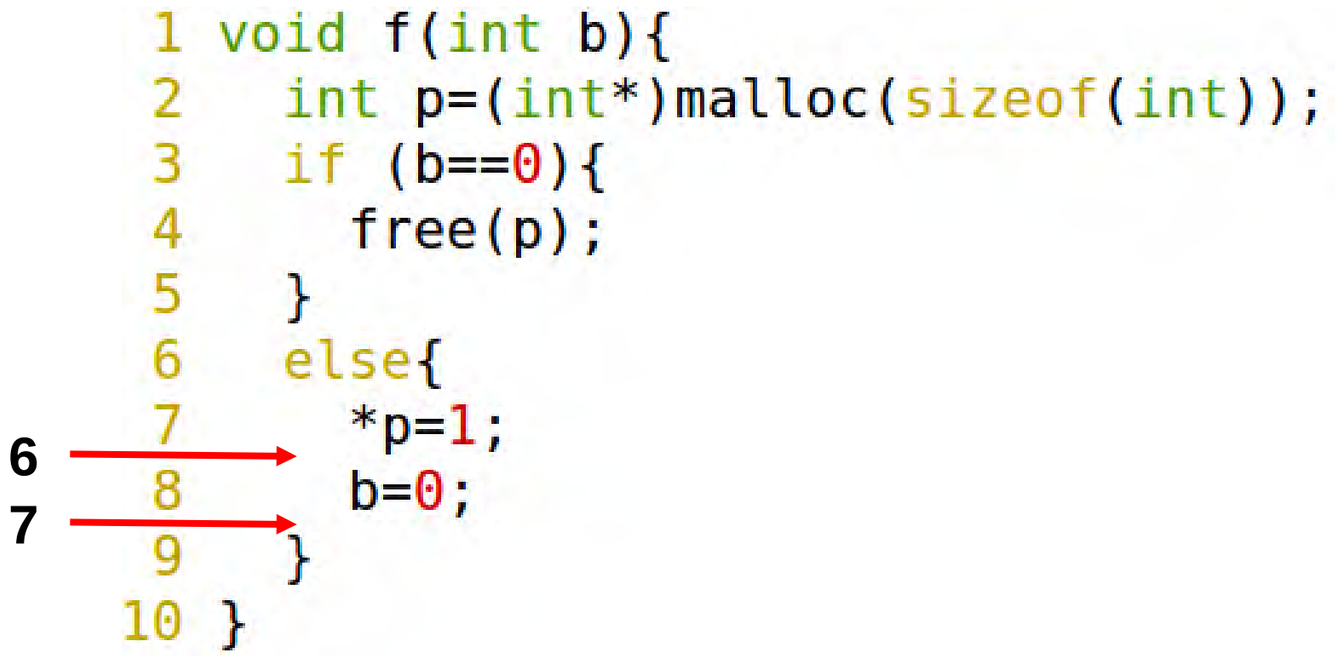
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

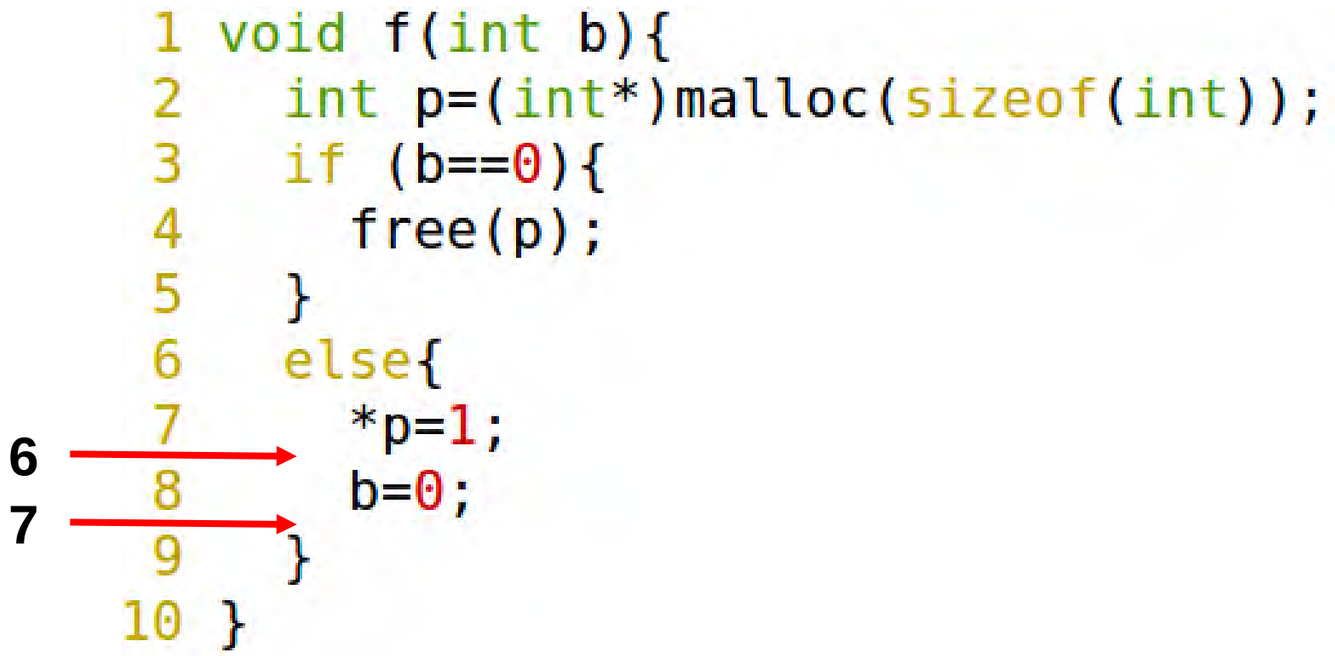
内存泄漏缺陷修复



问题的提出——示例

程序崩溃缺陷修复

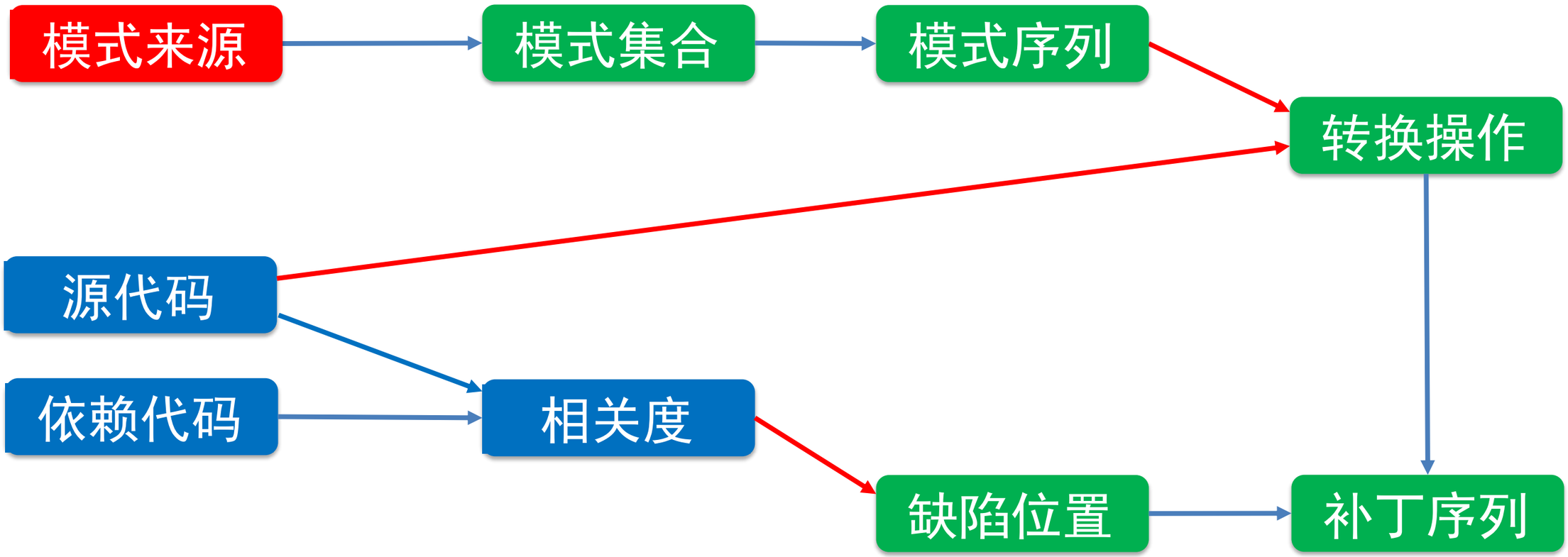
内存泄漏缺陷修复



内存泄漏缺陷的修复步骤

程序崩溃缺陷修复

内存泄漏缺陷修复



基于模式的缺陷定位——正确性

程序崩溃缺陷修复

内存泄漏缺陷修复

- 修复在分配之后
- 没有双重释放
- 释放后没有使用

- 数据流分析

- 检测内存泄漏
- 一个内存泄漏可能有多个修复点，但不保证完整

- 主要挑战：保证修复正确性，产生尽可能多的修复

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

Must-分配

分配

释放

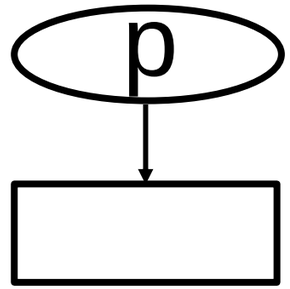
May-释放

使用

May-使用

```
1  void f(int b){
2    int p=(int*)malloc(sizeof(int));
3    if (b==0){
4      free(p);
5    }
6    else{
7      *p=1;
8      b=0;
9    }
10 }
```

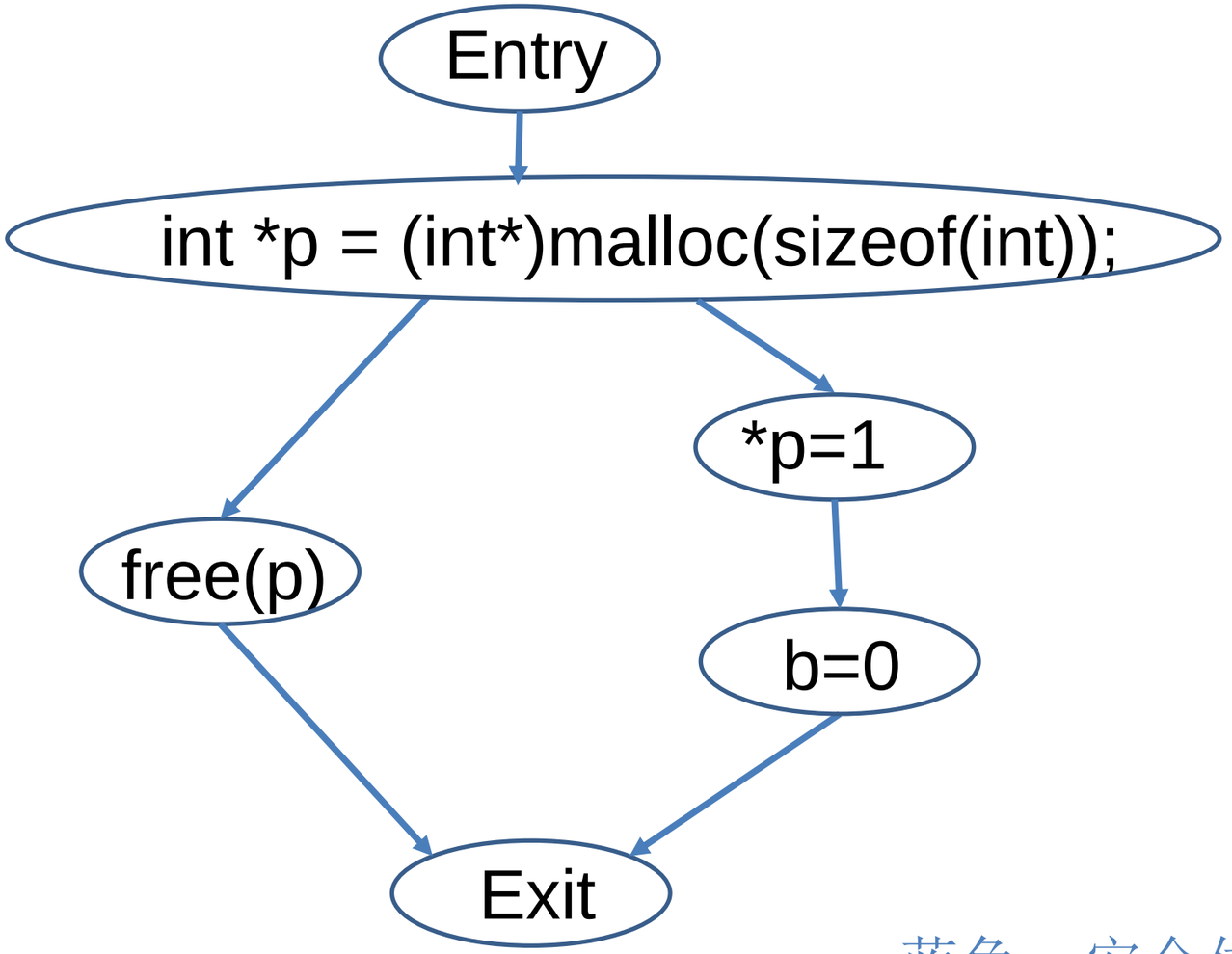
指针指向图



基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析



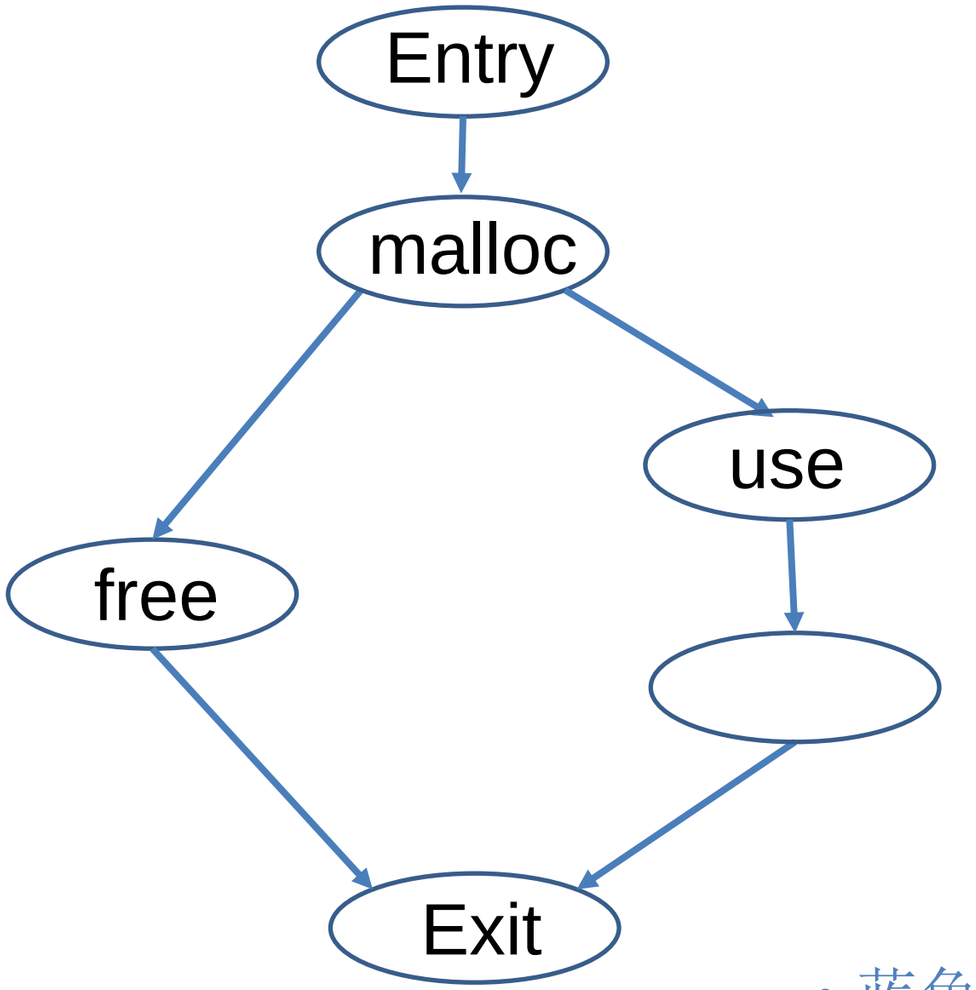
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

- 数据流分析

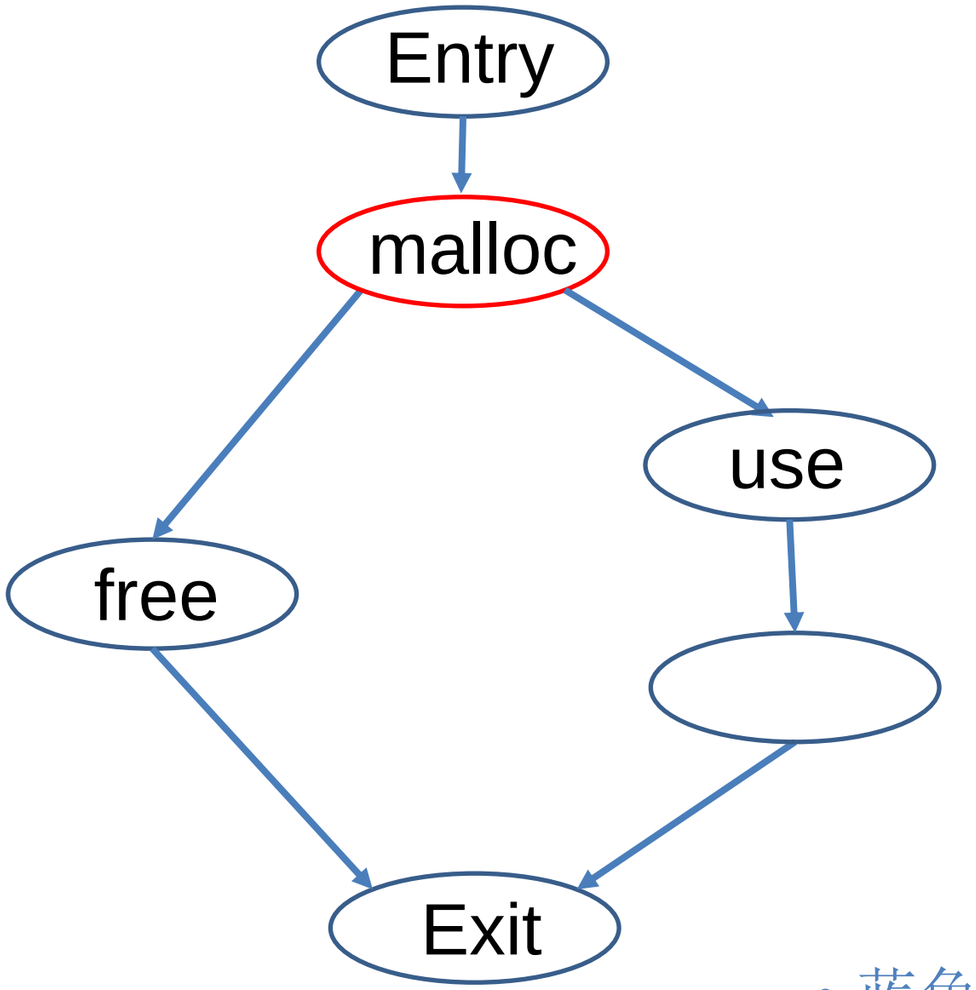


- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析

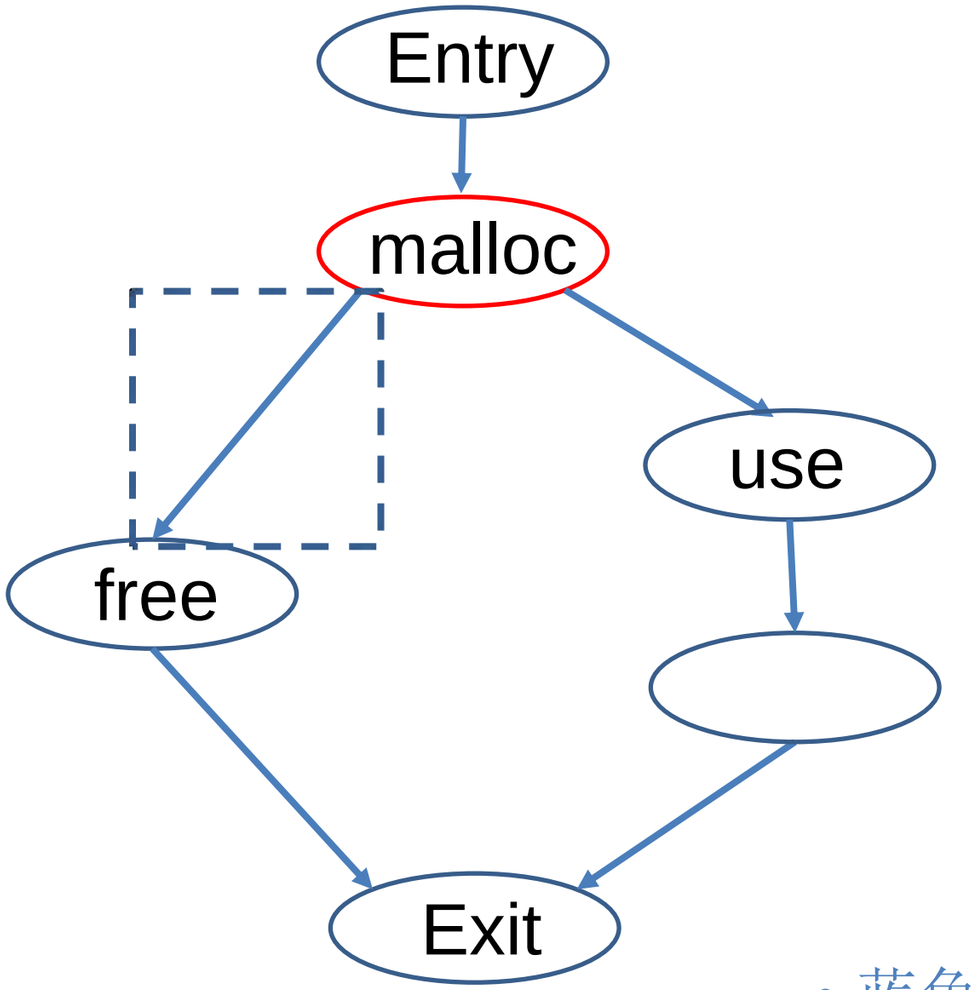


- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析



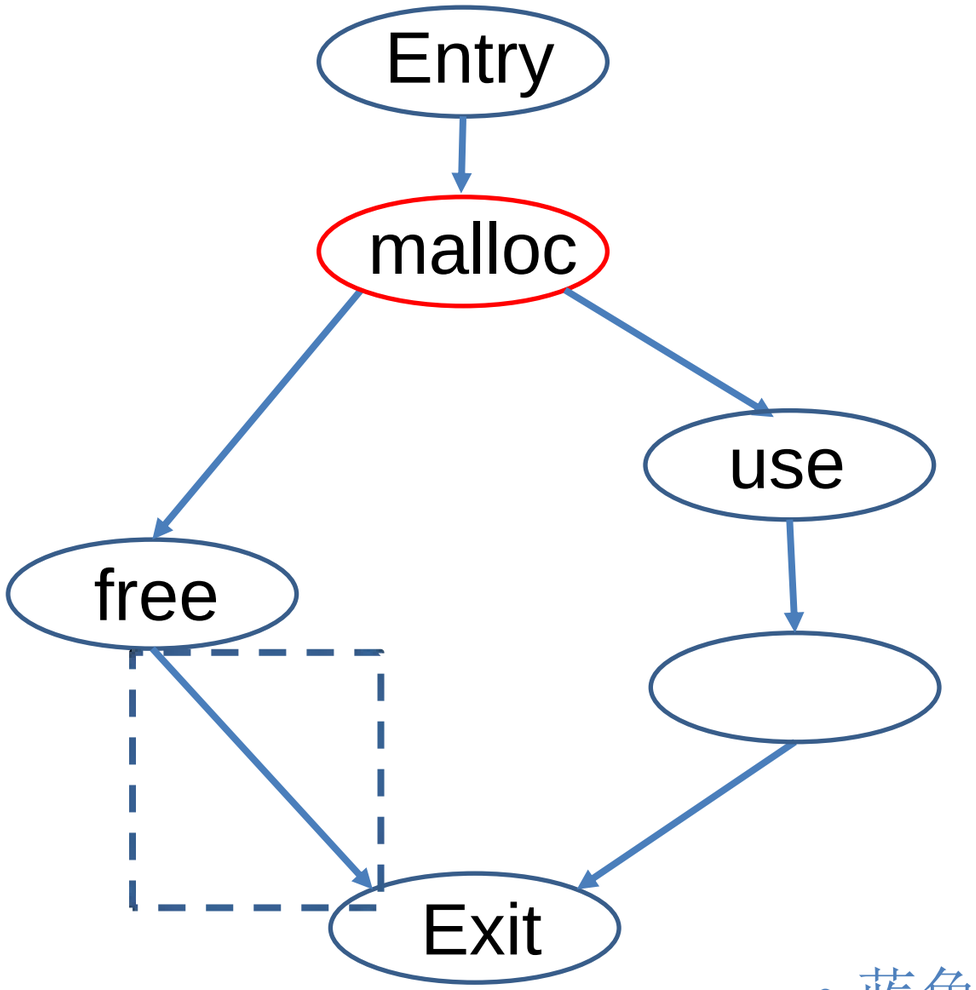
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

- 数据流分析

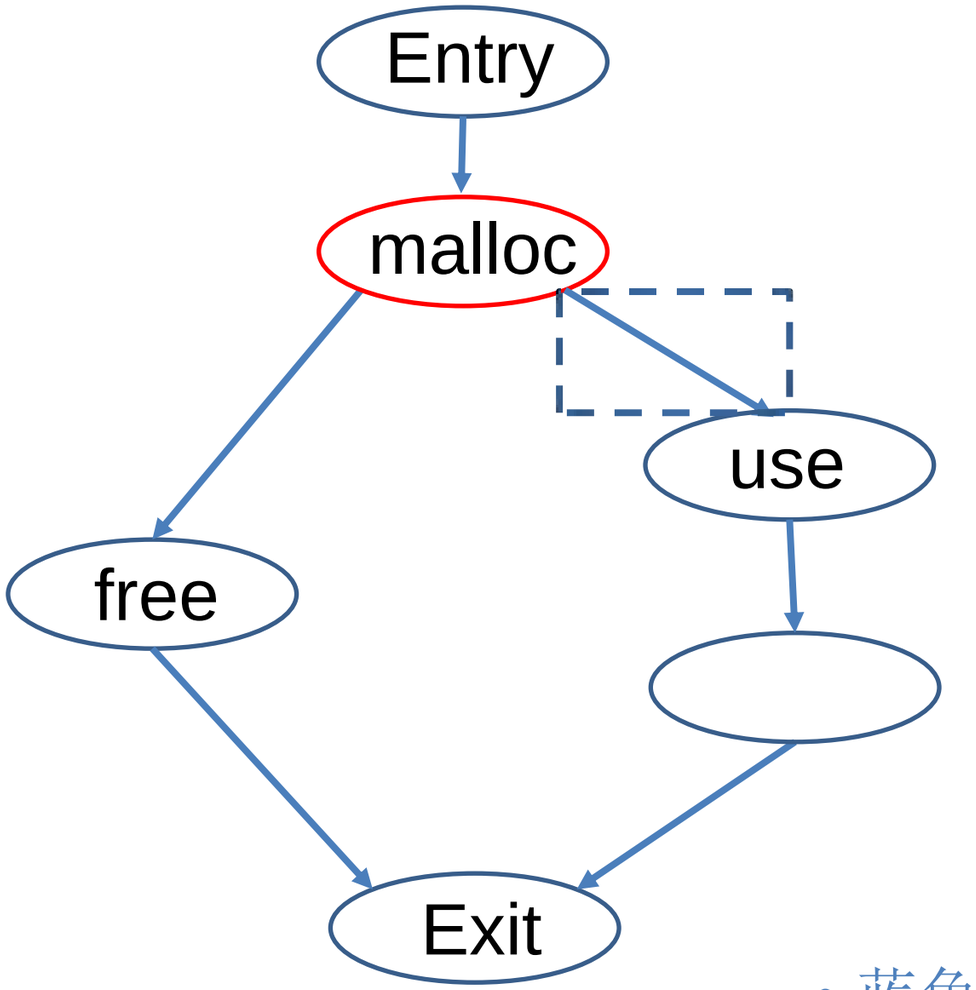


- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析

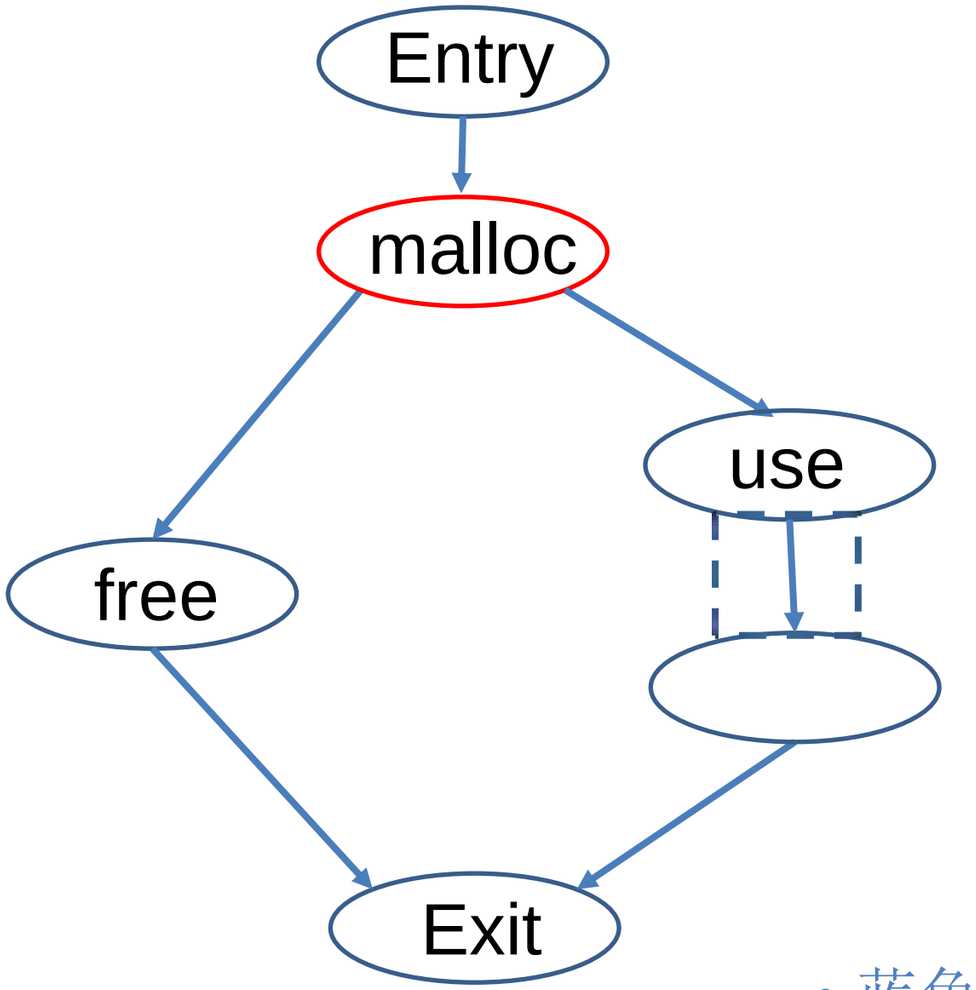


- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析

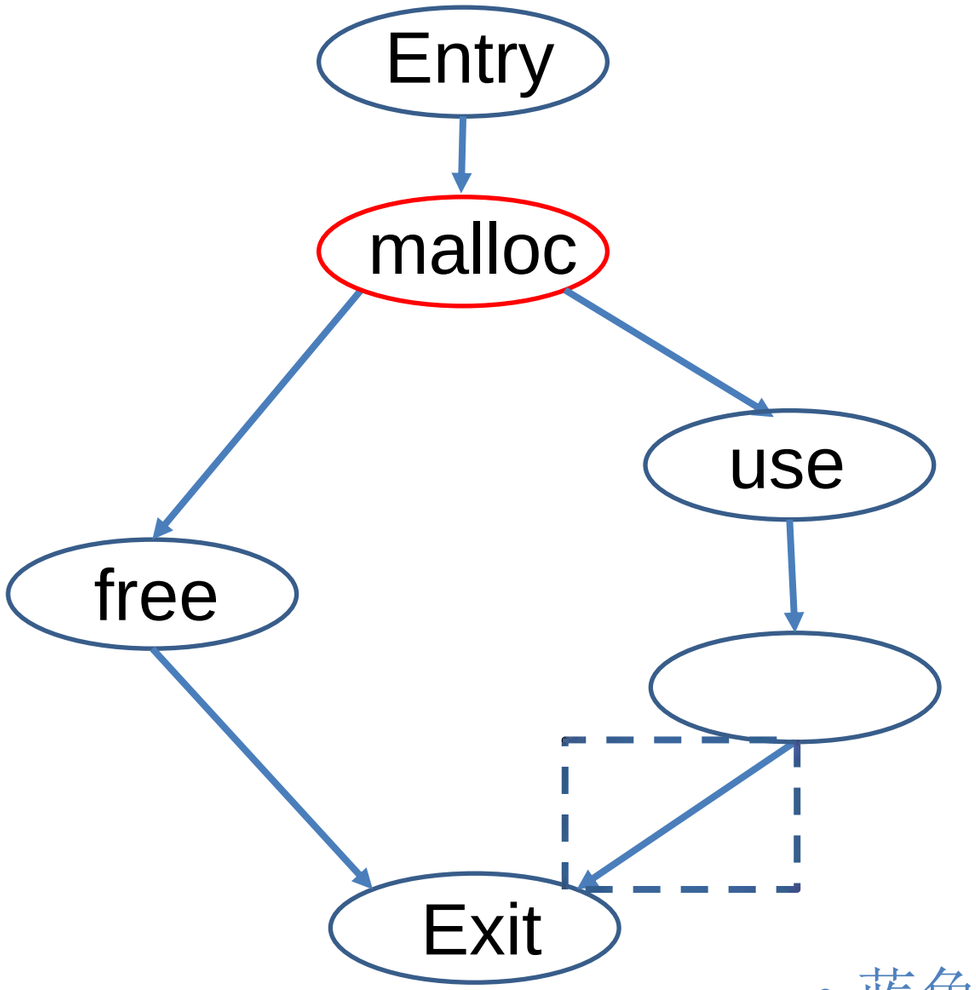


- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析



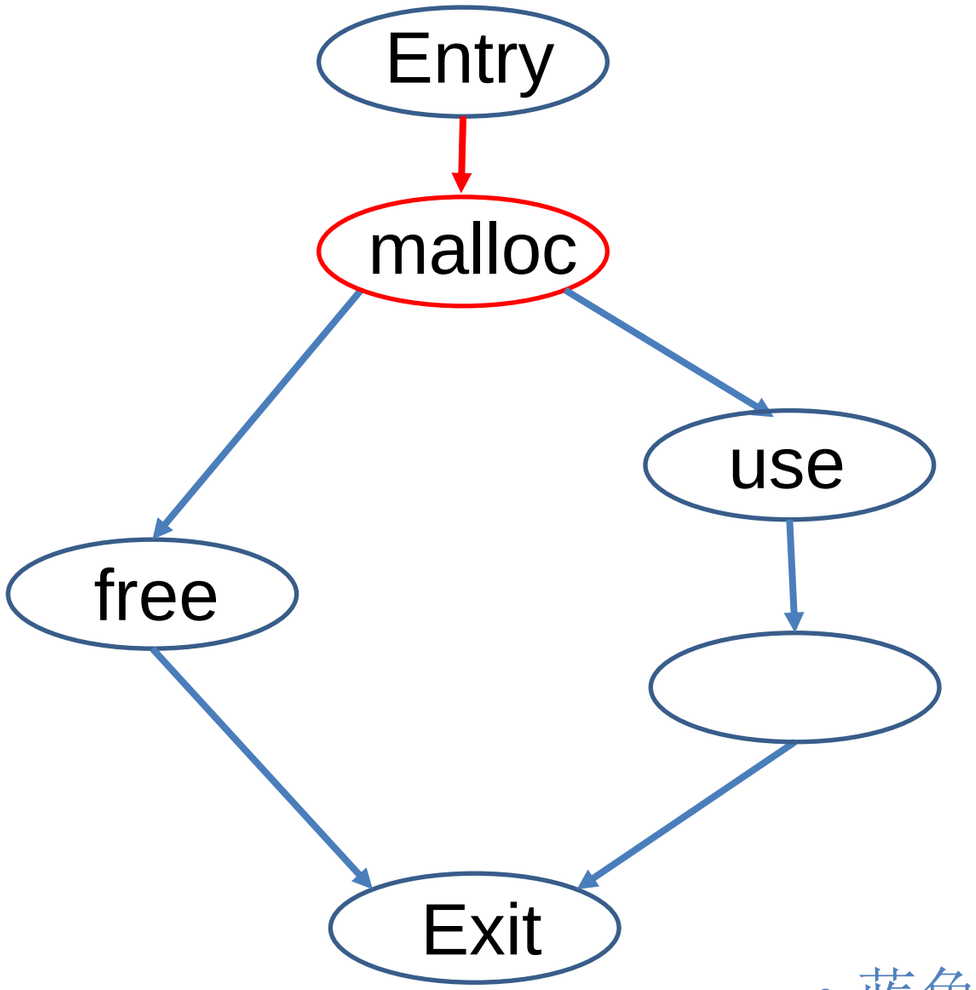
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

- 数据流分析



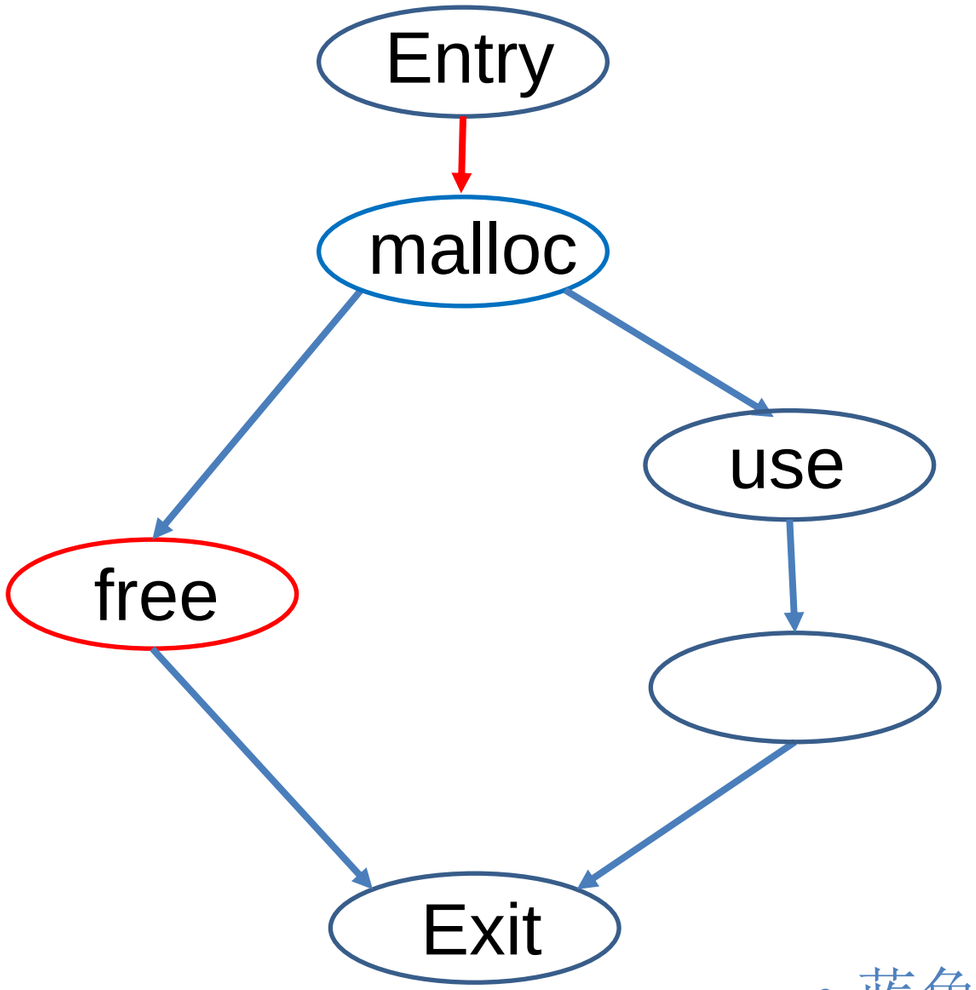
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

- 数据流分析



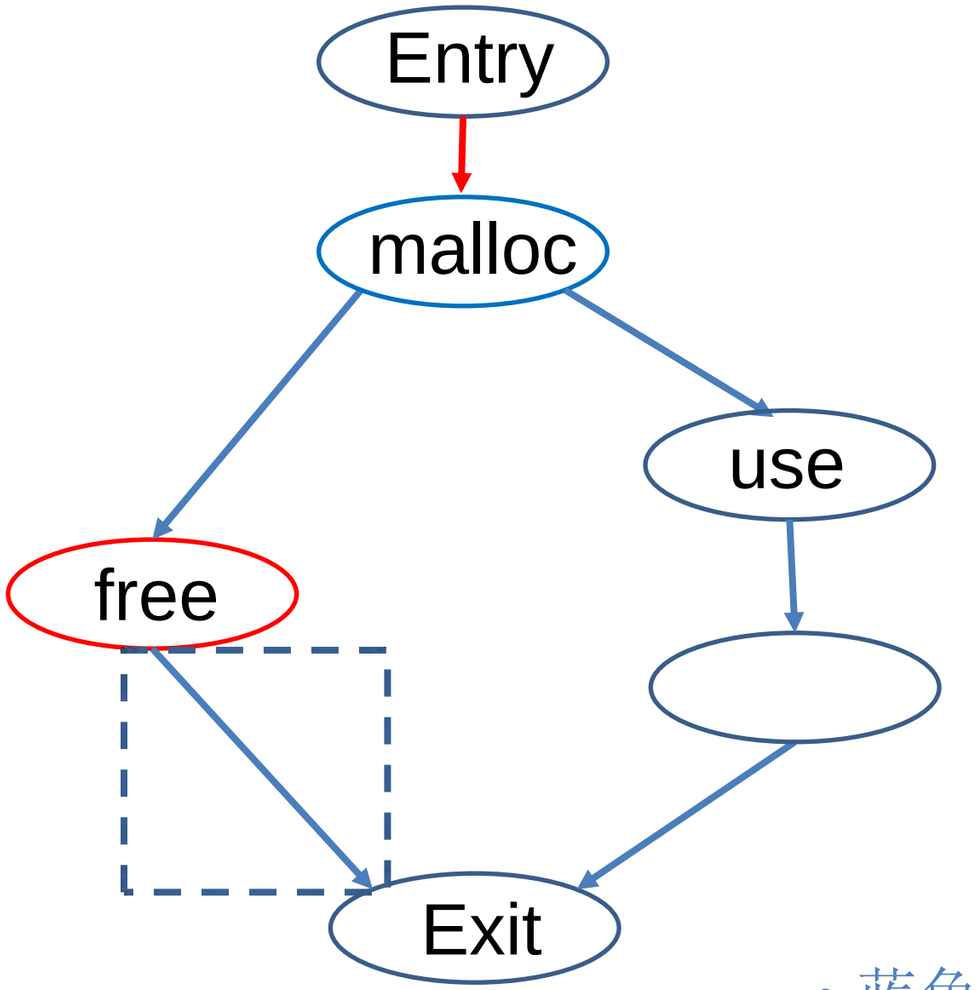
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

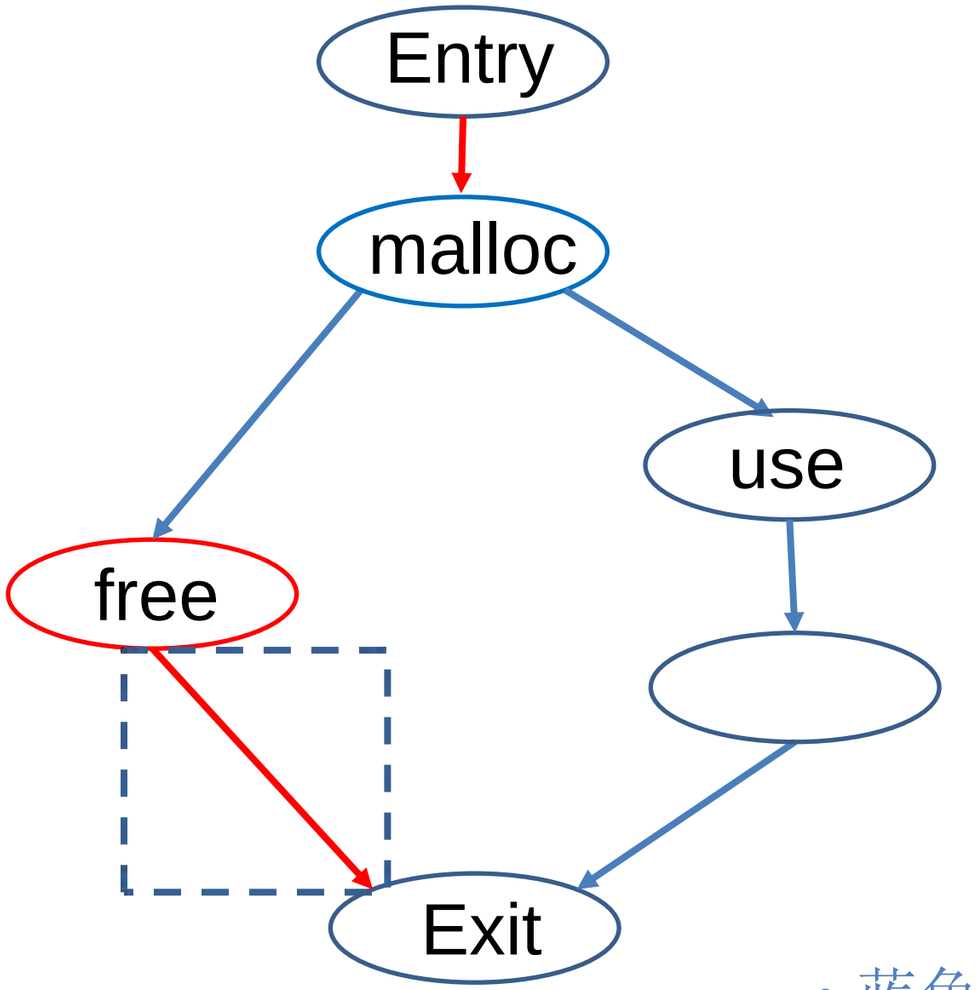
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

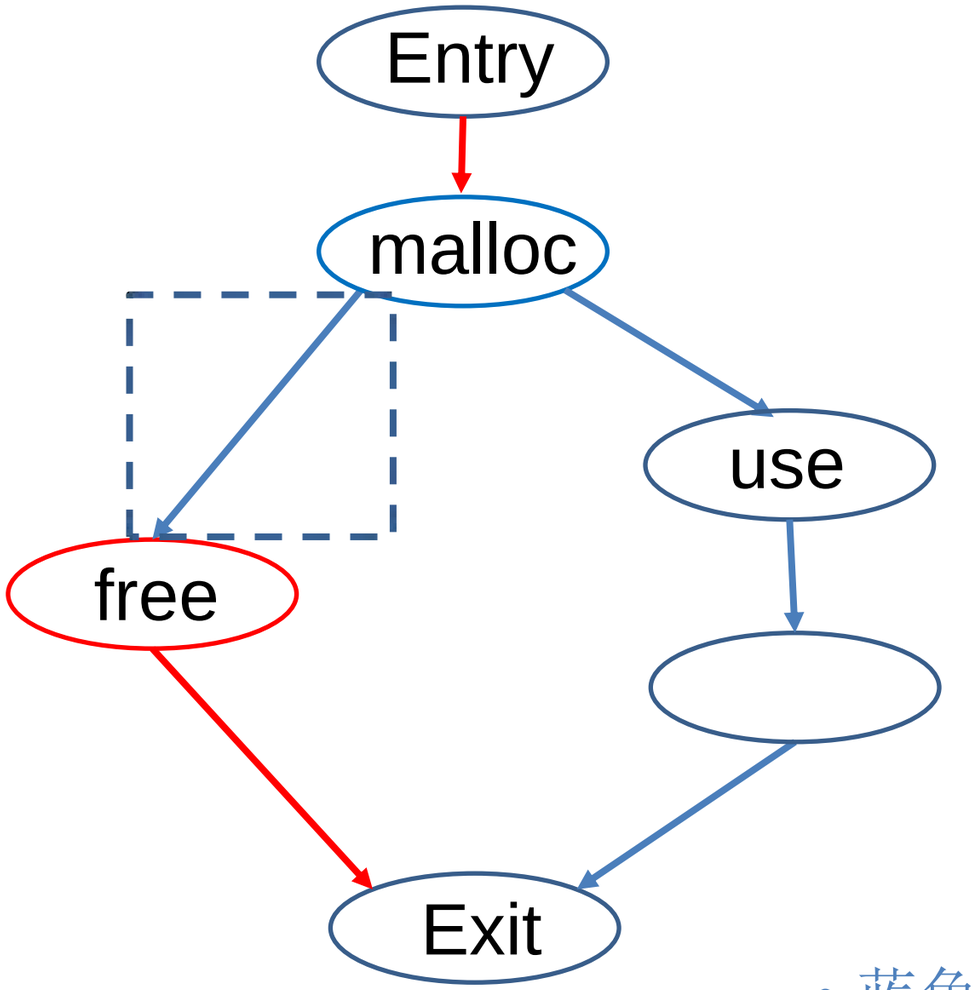
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

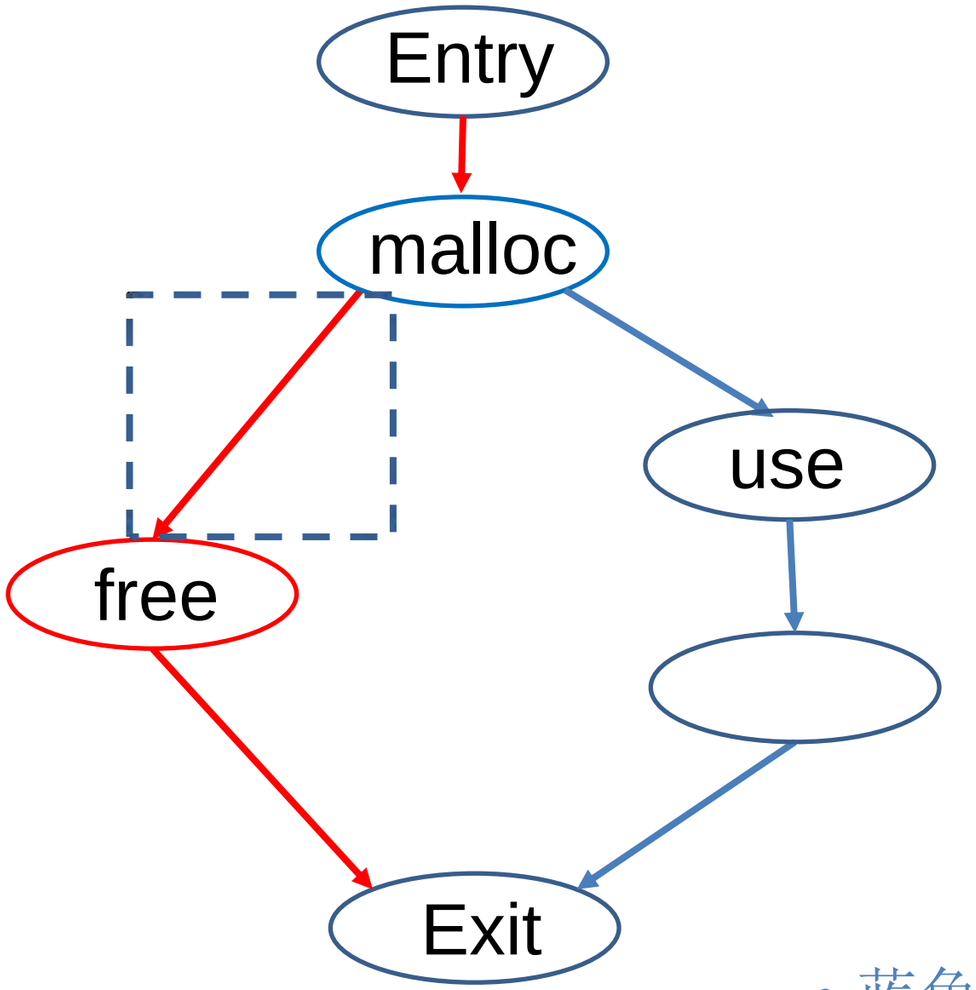
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

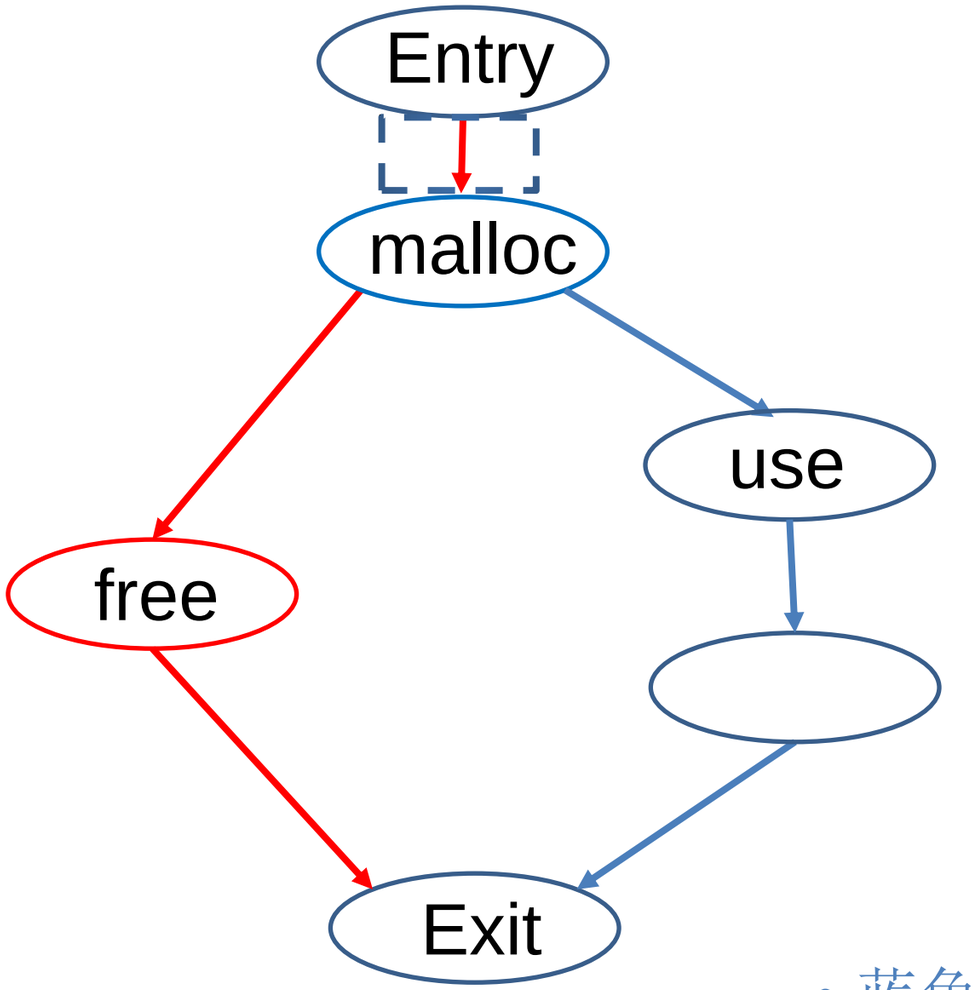
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

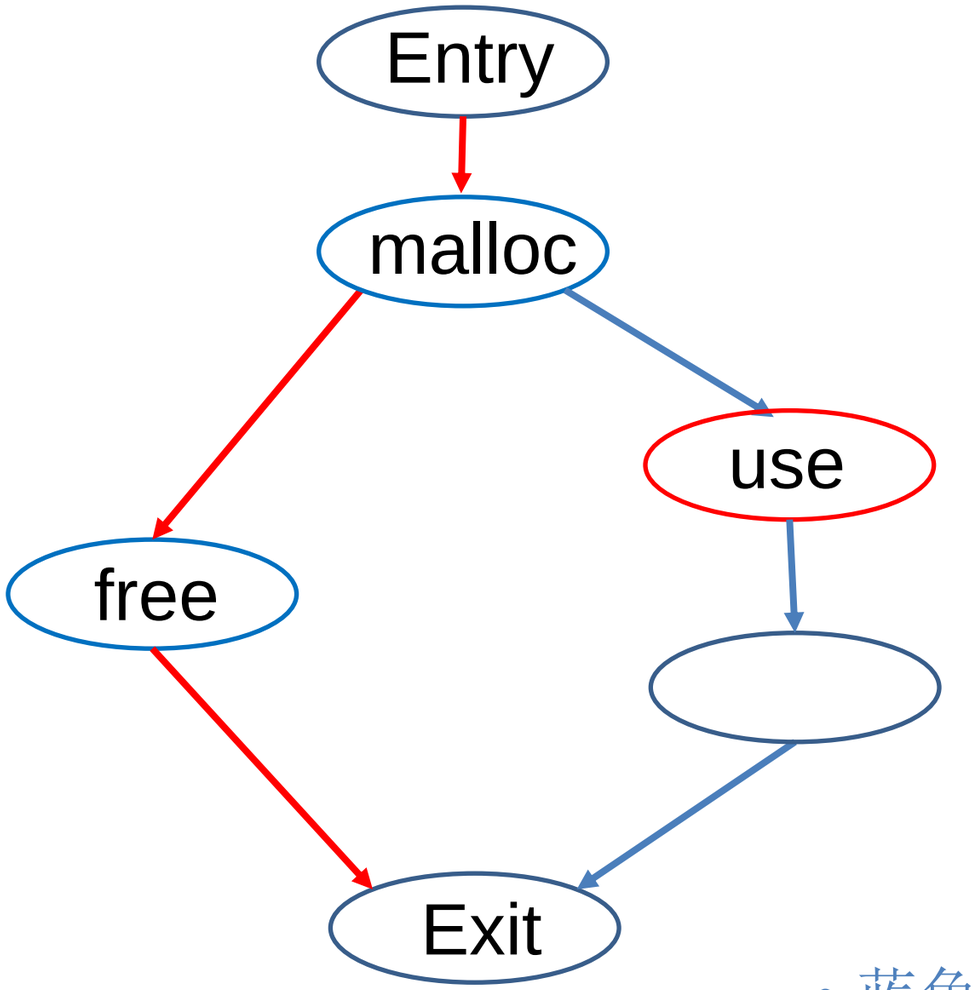
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

- 数据流分析

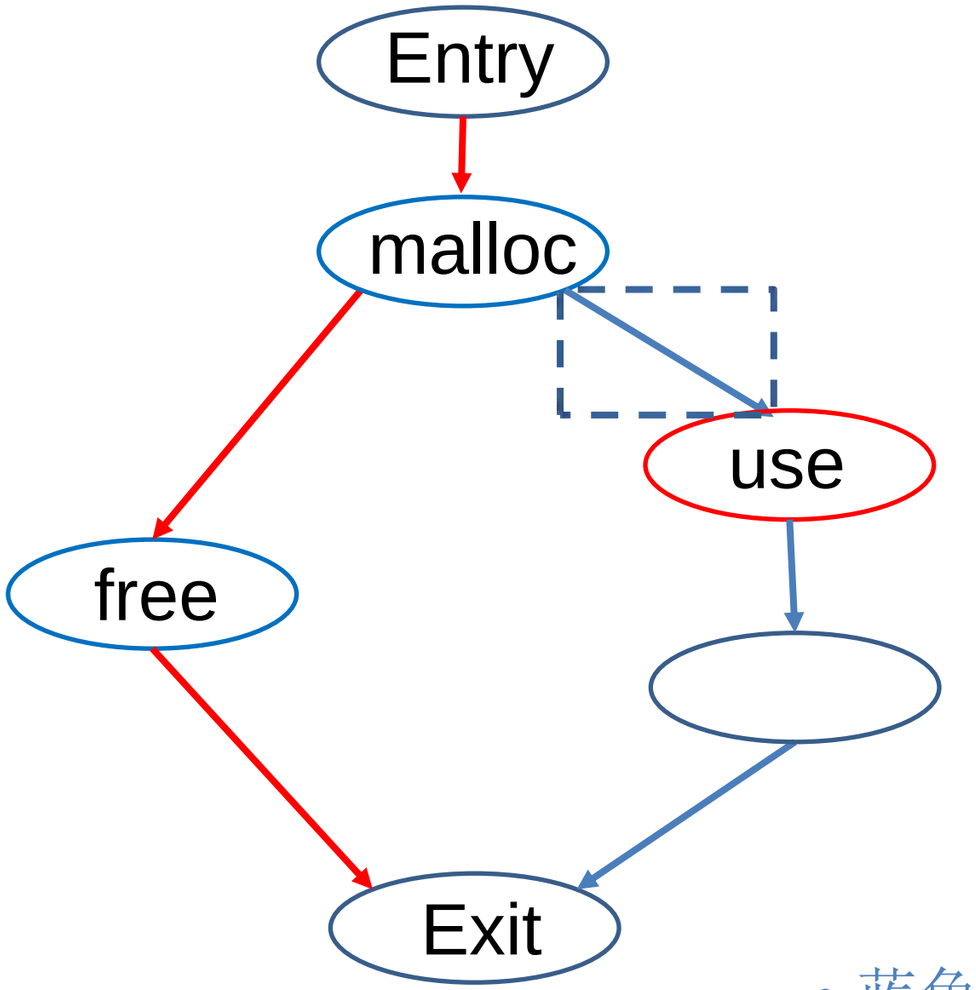


- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复　内存泄漏缺陷修复

- 数据流分析



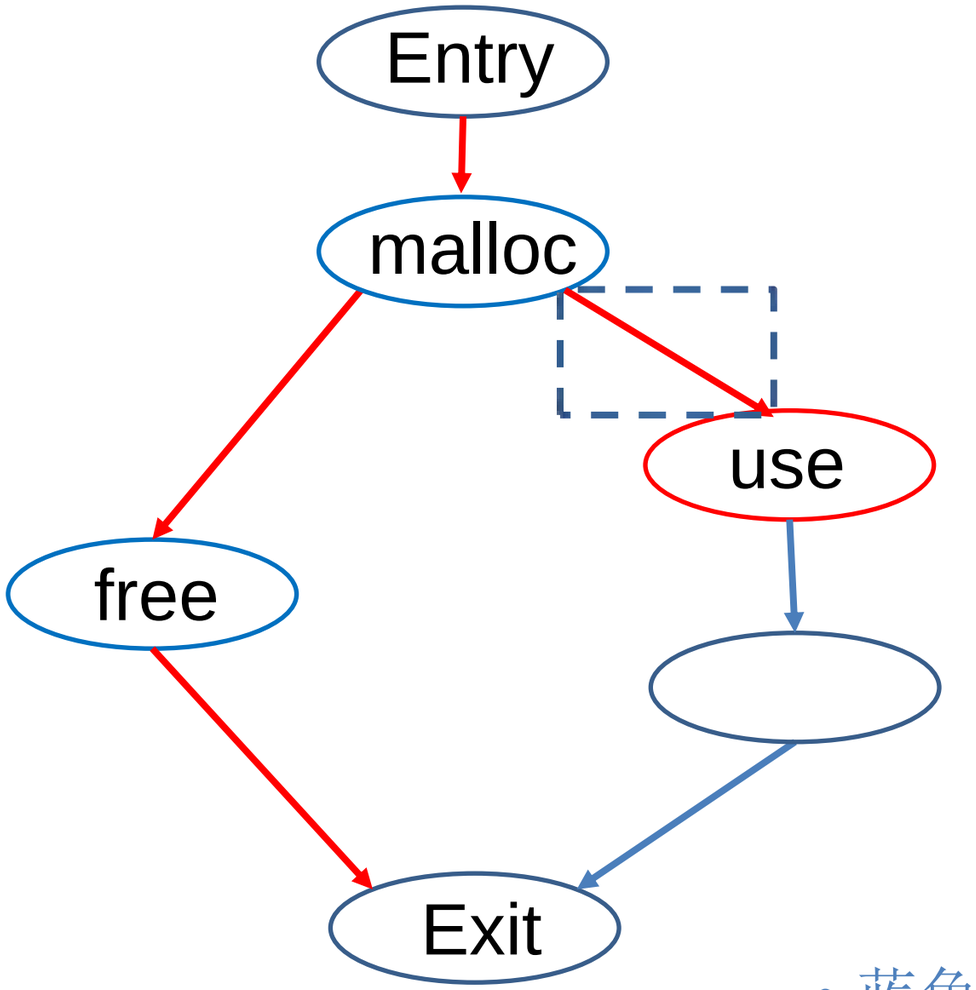
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

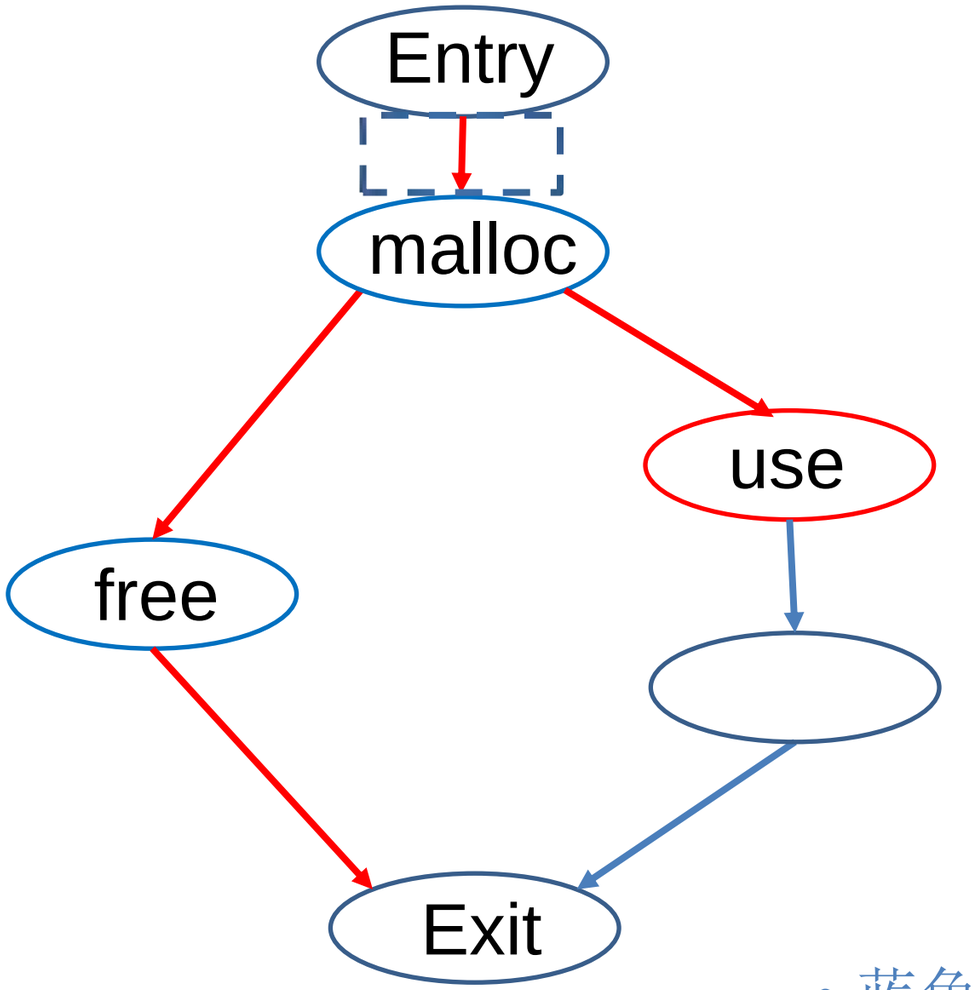
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

- 数据流分析



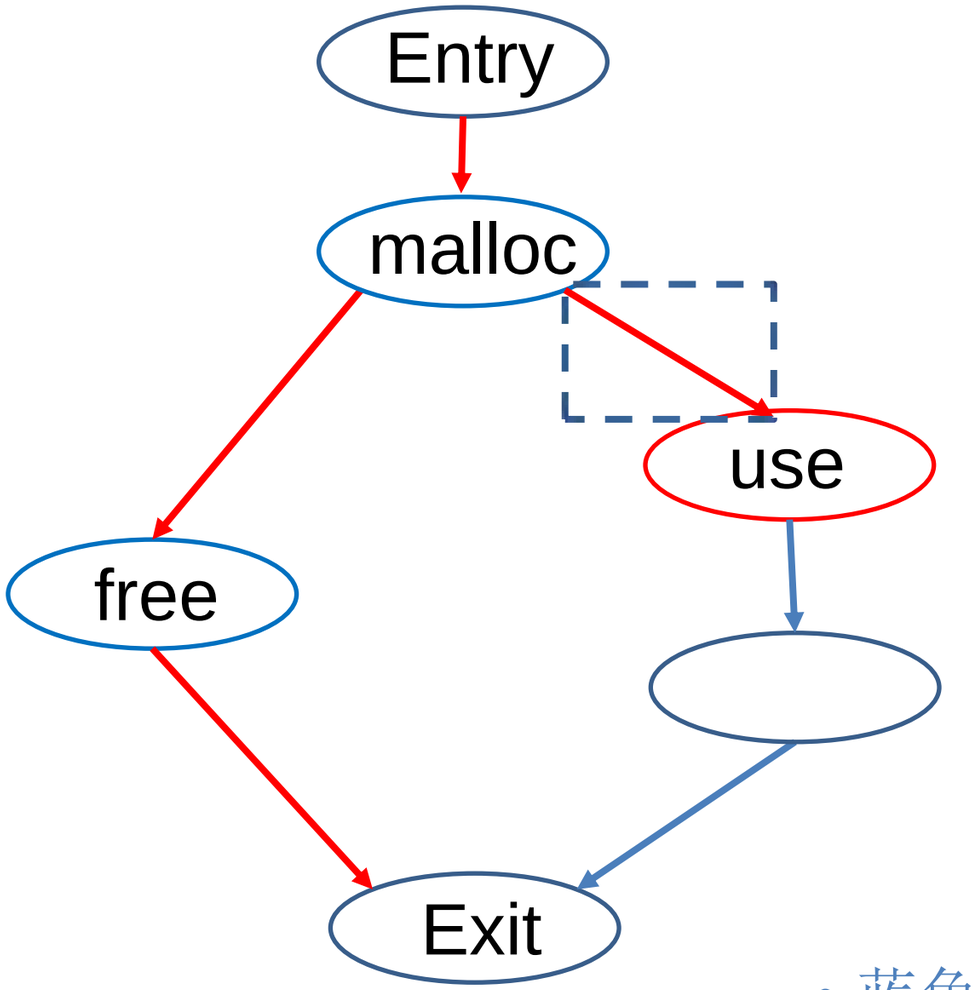
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

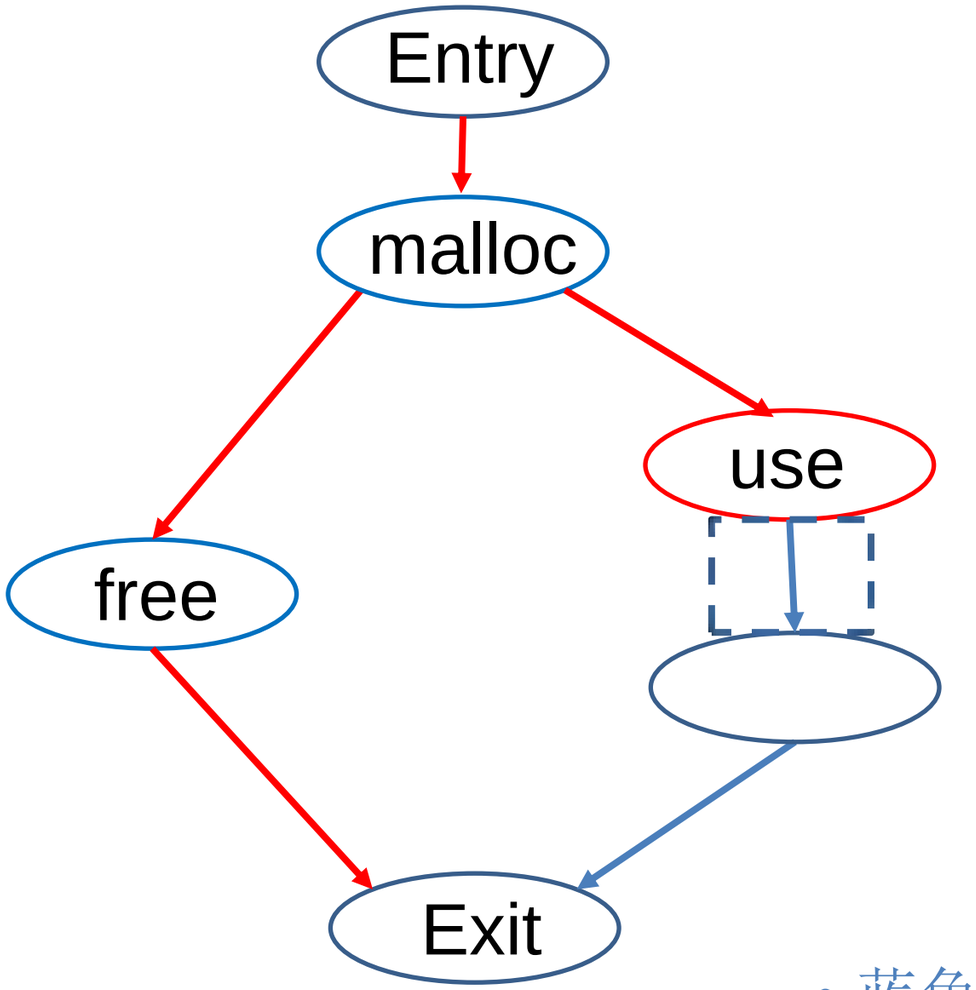
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

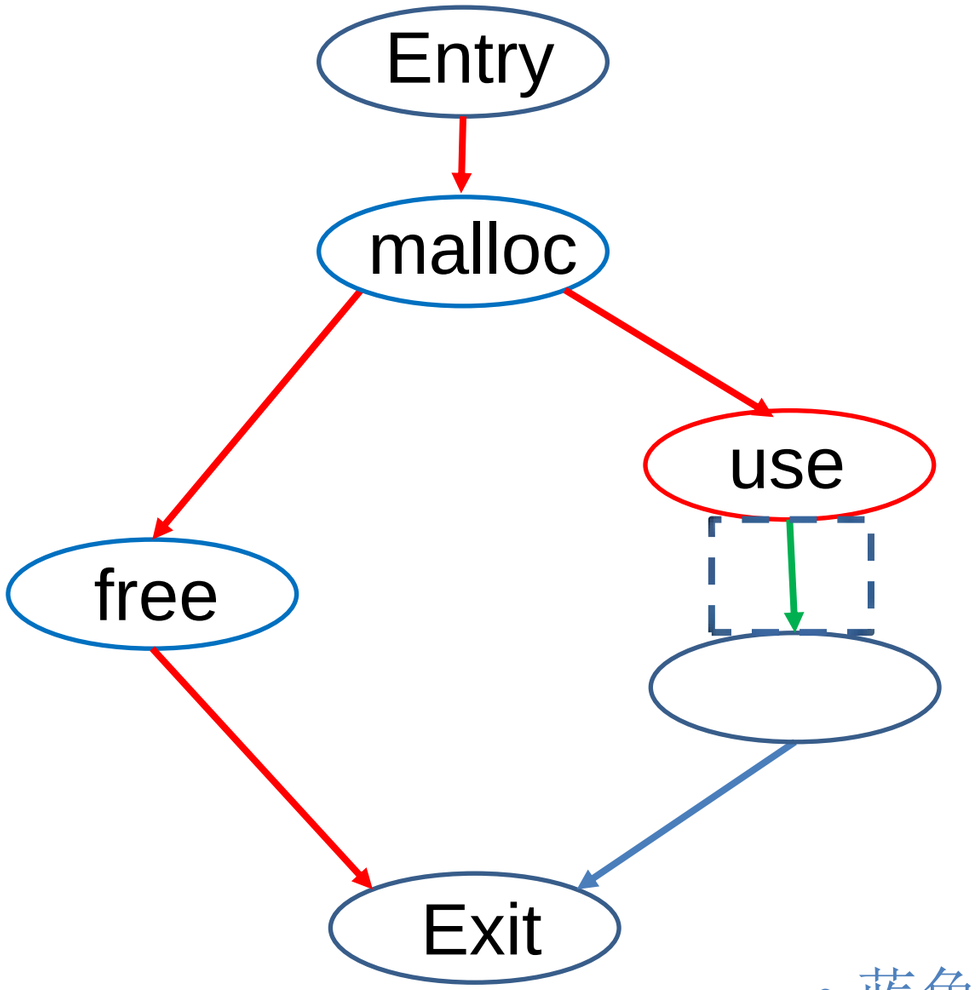
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

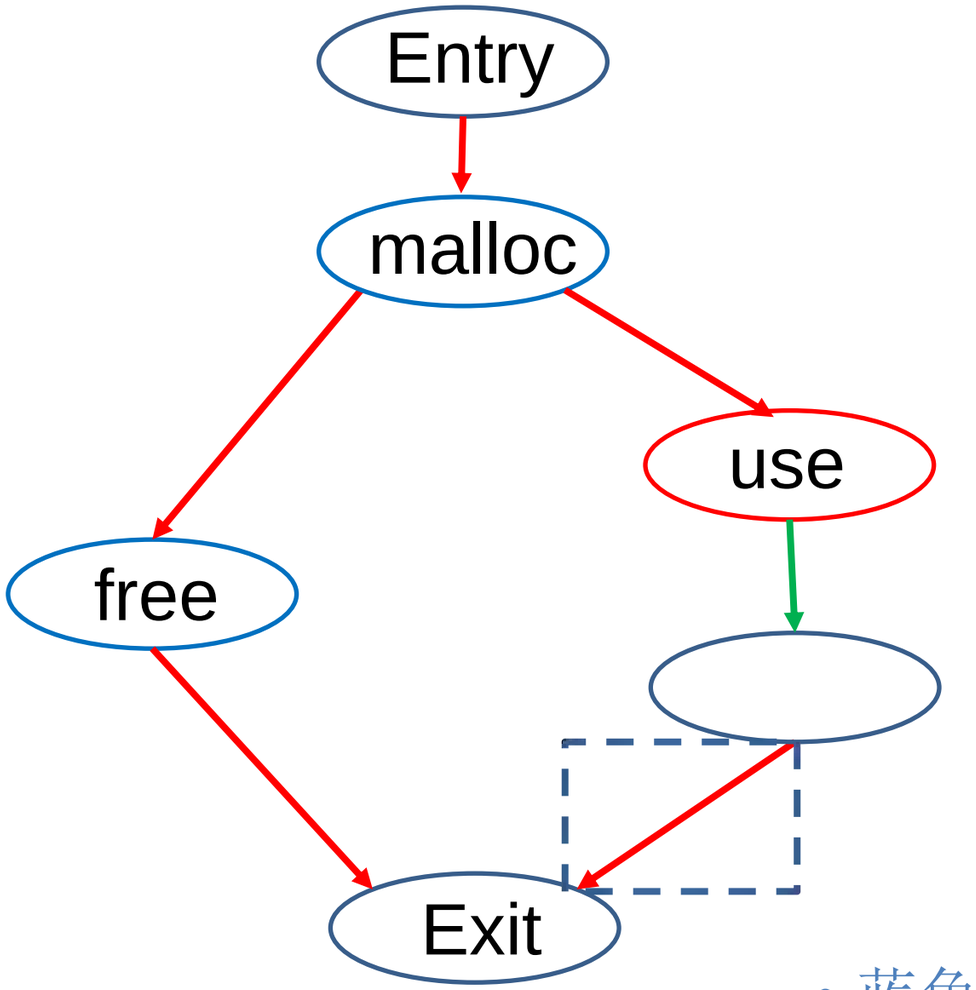
- 数据流分析



- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

- 数据流分析



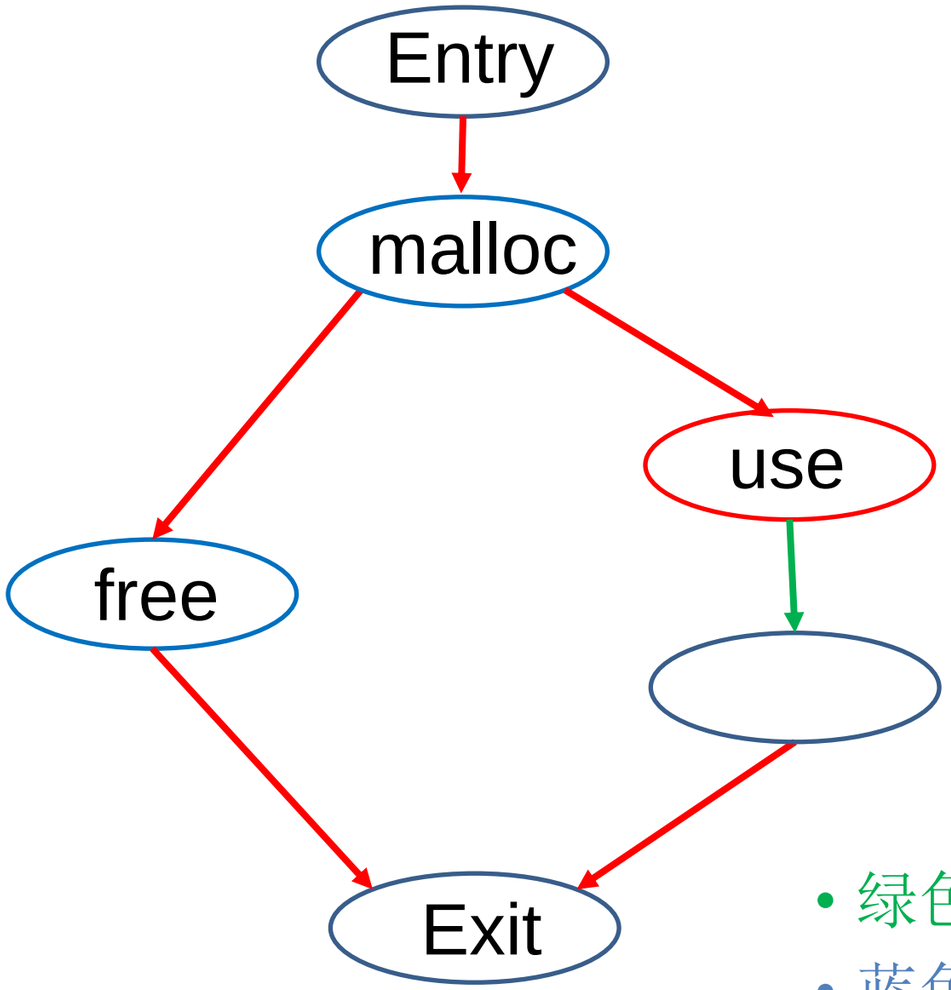
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

程序崩溃缺陷修复

内存泄漏缺陷修复

- 数据流分析



- 绿色：修复位置
- 蓝色：安全位置
- 红色：不安全位置

基于模式的缺陷定位

- 复杂情况下的缺陷定位
  - ✓过程间分析：构建摘要

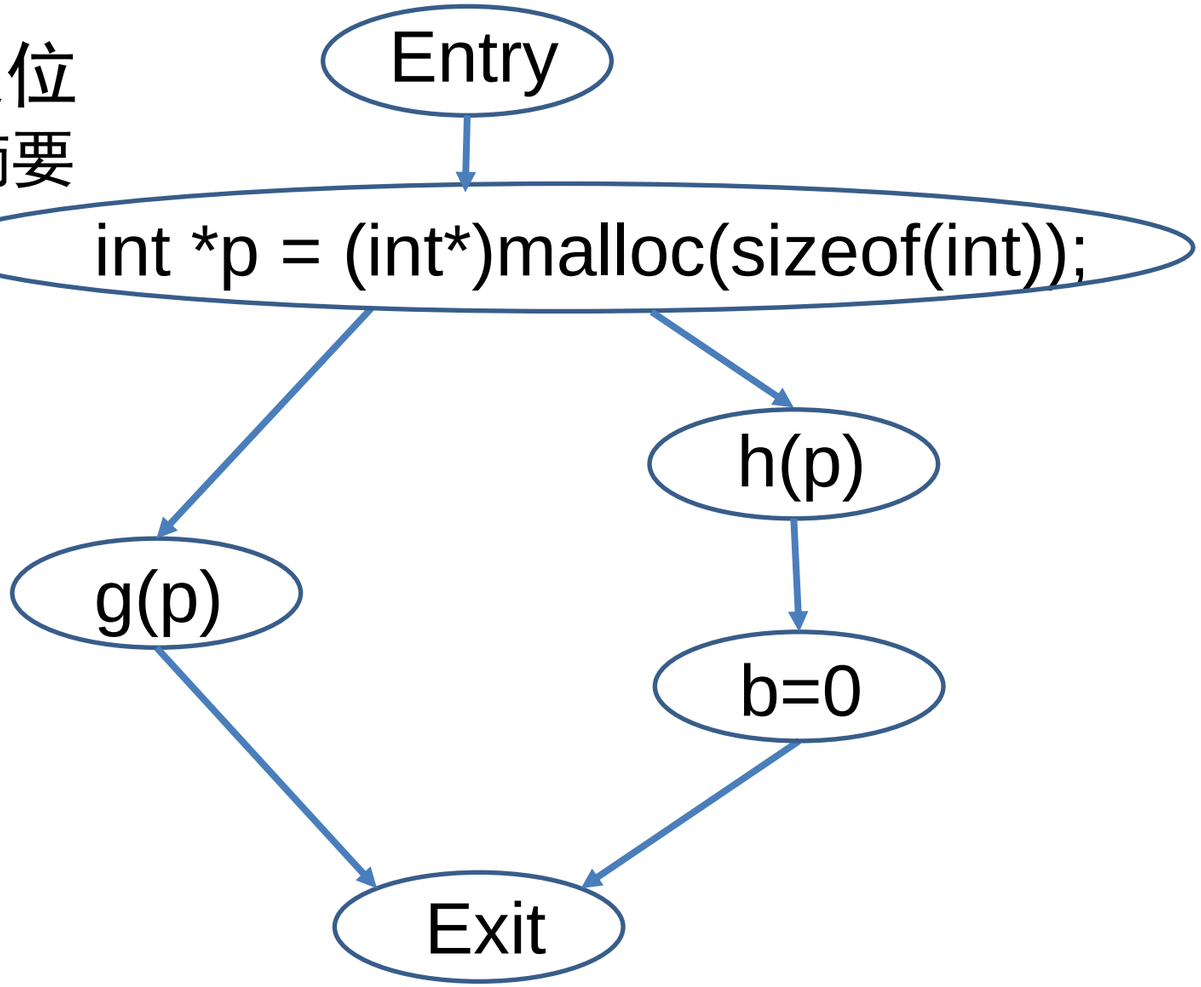
```
void g(int *p){
  free(p);
}

void h(int *p){
  *p=1;
}
```

```
1  void f(int b){
2    int p=(int*)malloc(sizeof(int));
3    if (b==0){
4      g(p);
5    }
6    else{
7      h(p);
8      b=0;
9    }
10 }
```

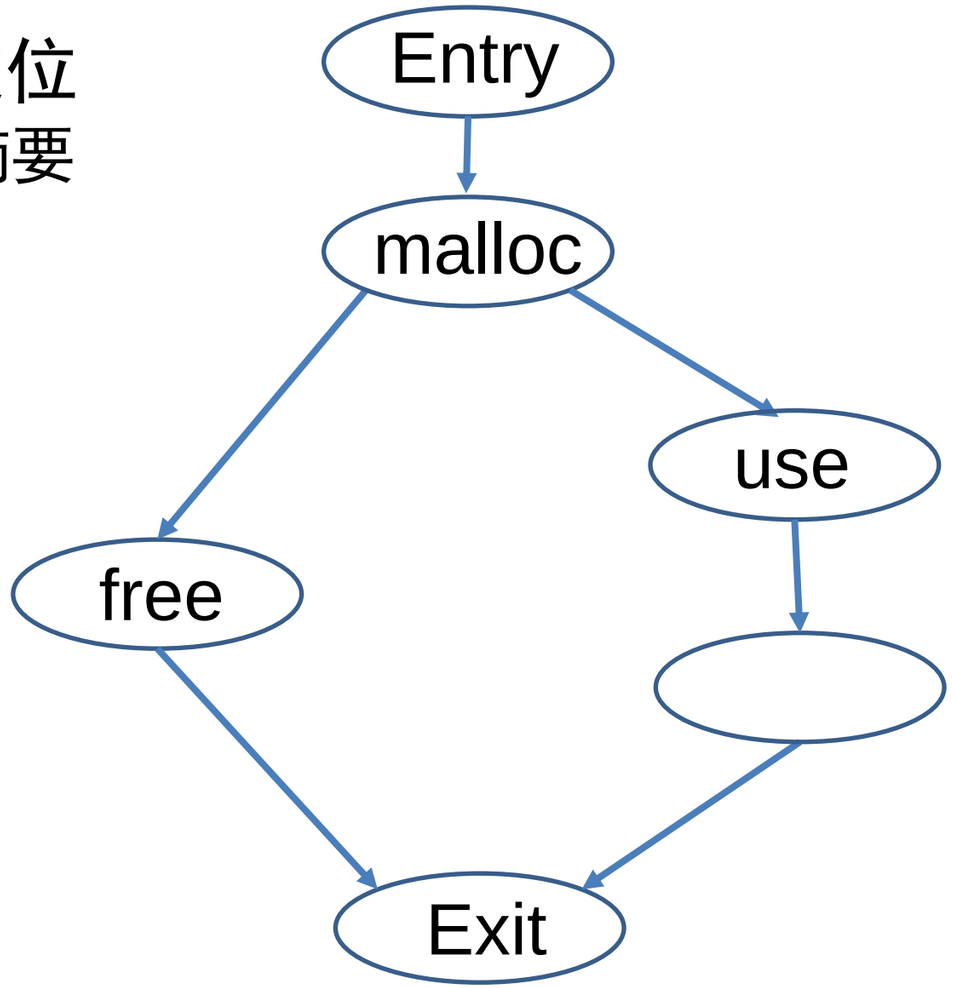
基于模式的缺陷定位

- 复杂情况下的缺陷定位
  - ✓过程间分析：构建摘要



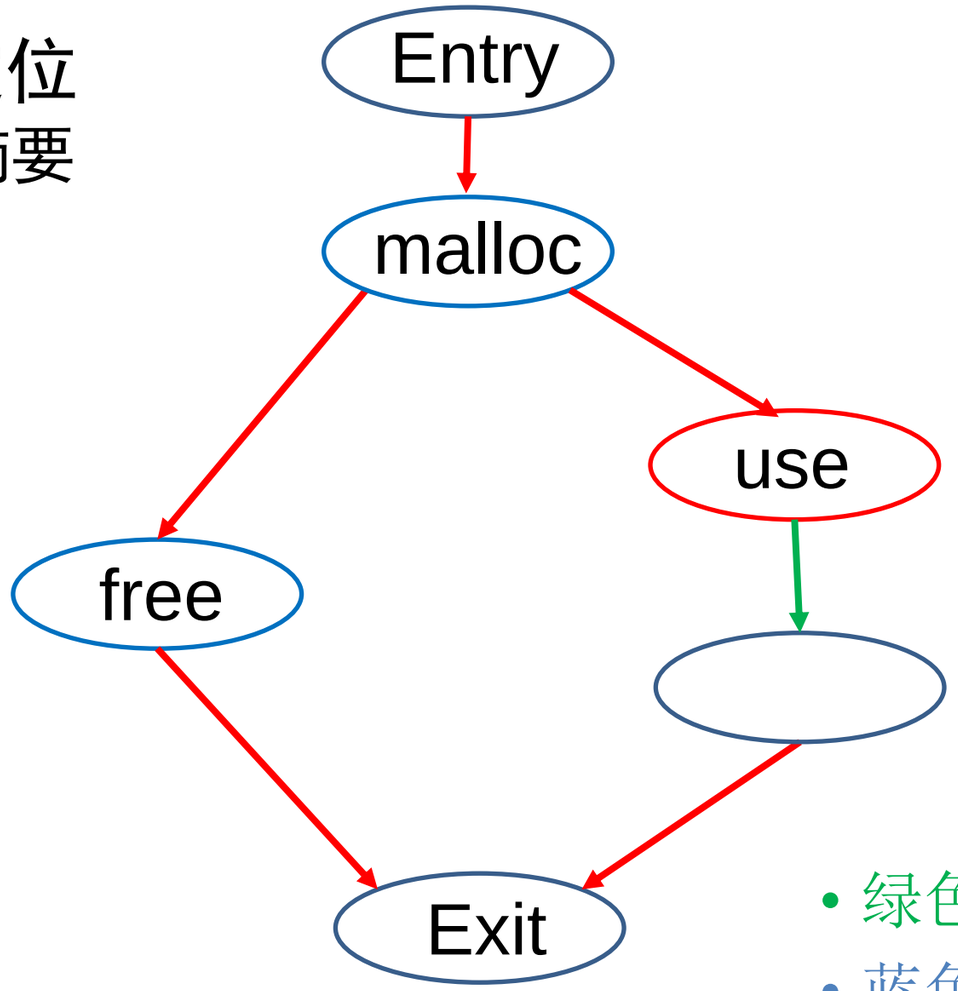
基于模式的缺陷定位

- 复杂情况下的缺陷定位
  - ✓过程间分析：构建摘要



基于模式的缺陷定位

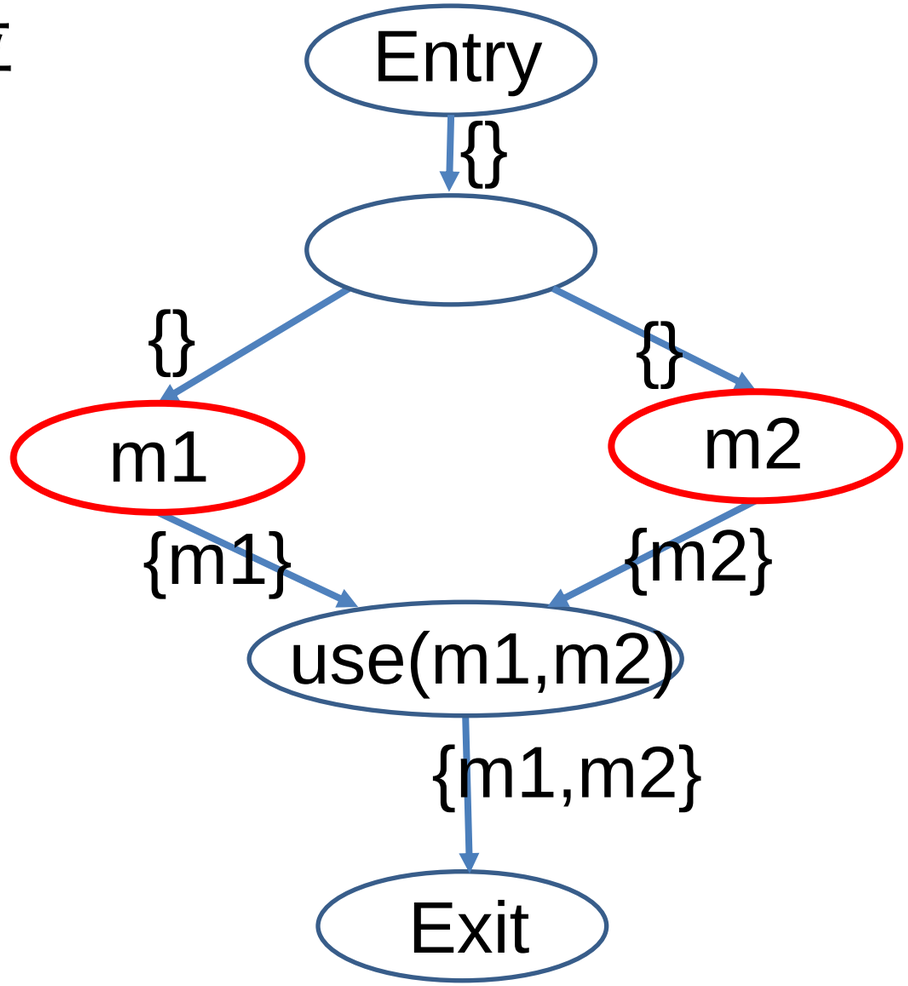
- 复杂情况下的缺陷定位
  - ✓过程间分析：构建摘要



- 绿色：修复位置
- 蓝色：安全位置
- 红色：不安全位置

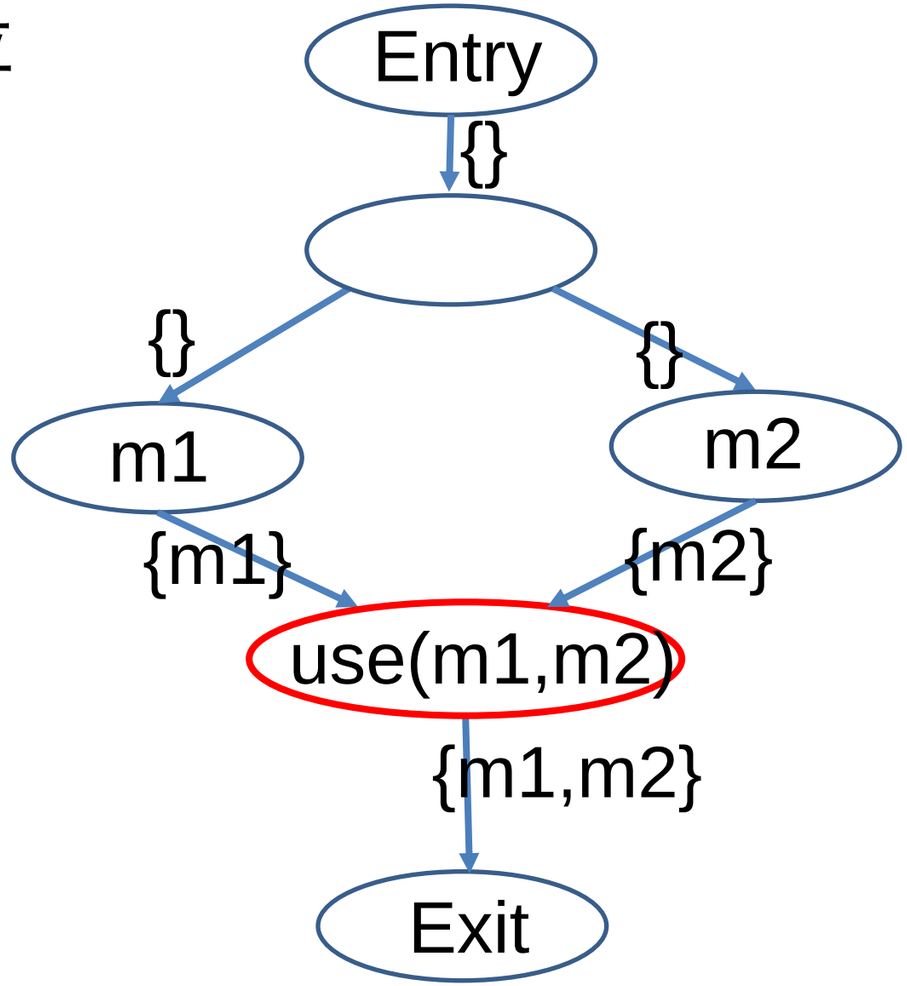
基于模式的缺陷定位

- 复杂情况下的缺陷定位
  - ✓ 多重分配



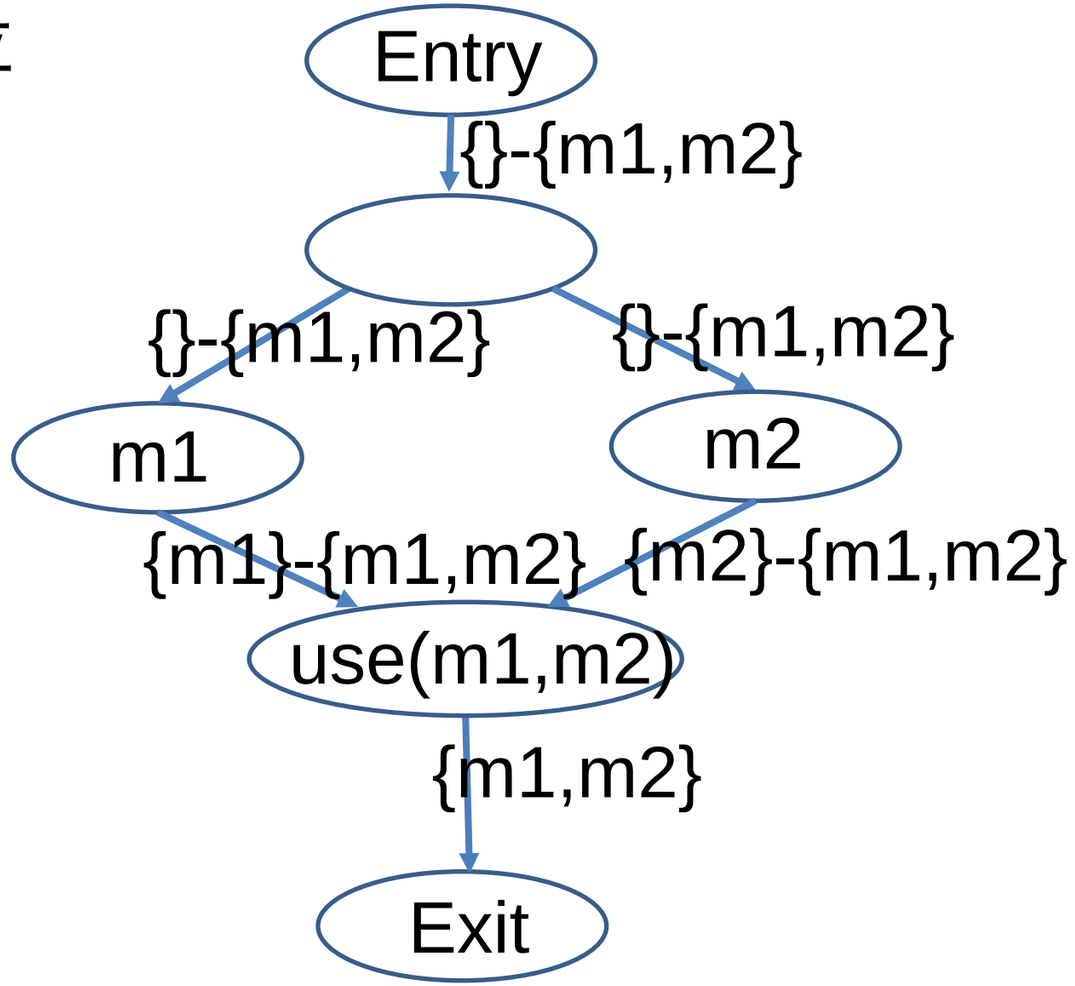
基于模式的缺陷定位

- 复杂情况下的缺陷定位
  - ✓ 多重分配



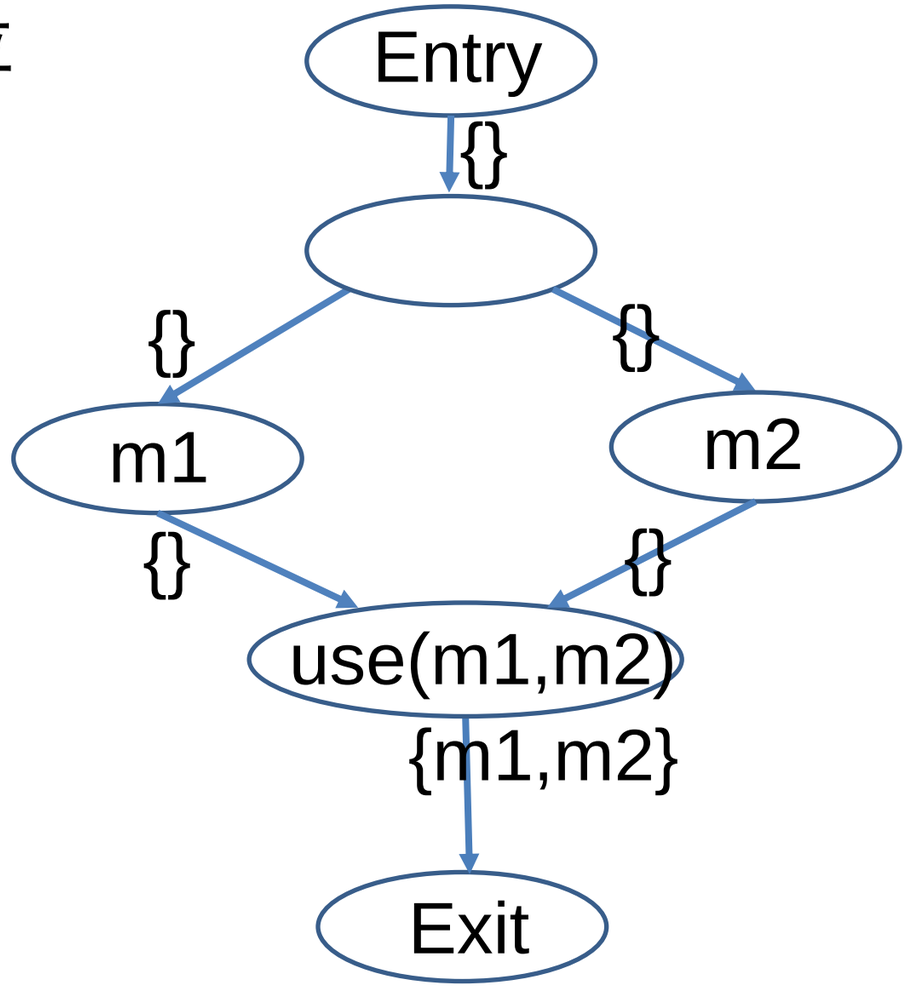
基于模式的缺陷定位

- 复杂情况下的缺陷定位
  - ✓ 多重分配



基于模式的缺陷定位

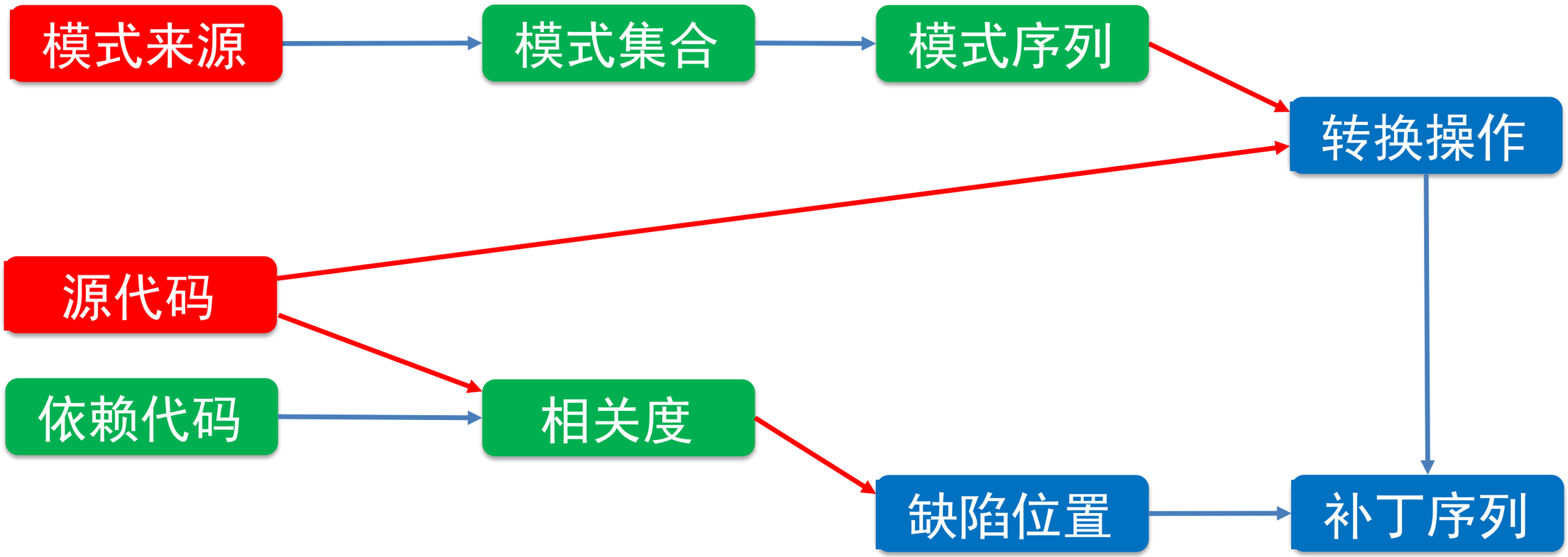
- 复杂情况下的缺陷定位
  - ✓多重分配



程序崩溃缺陷的修复步骤

程序崩溃缺陷修复

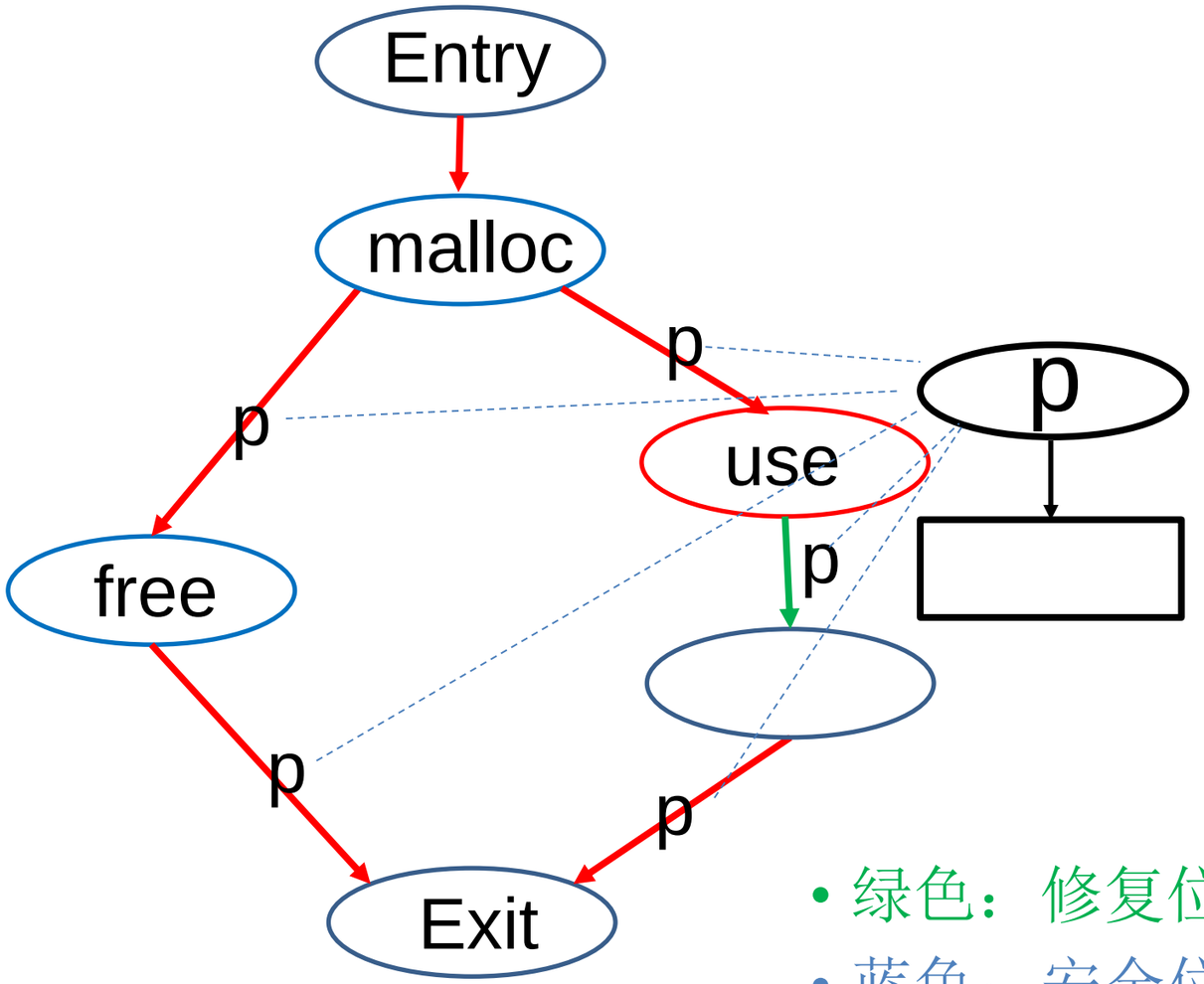
内存泄漏缺陷修复



模式应用

程序崩溃缺陷修复　内存泄漏缺陷修复

- 指针分析

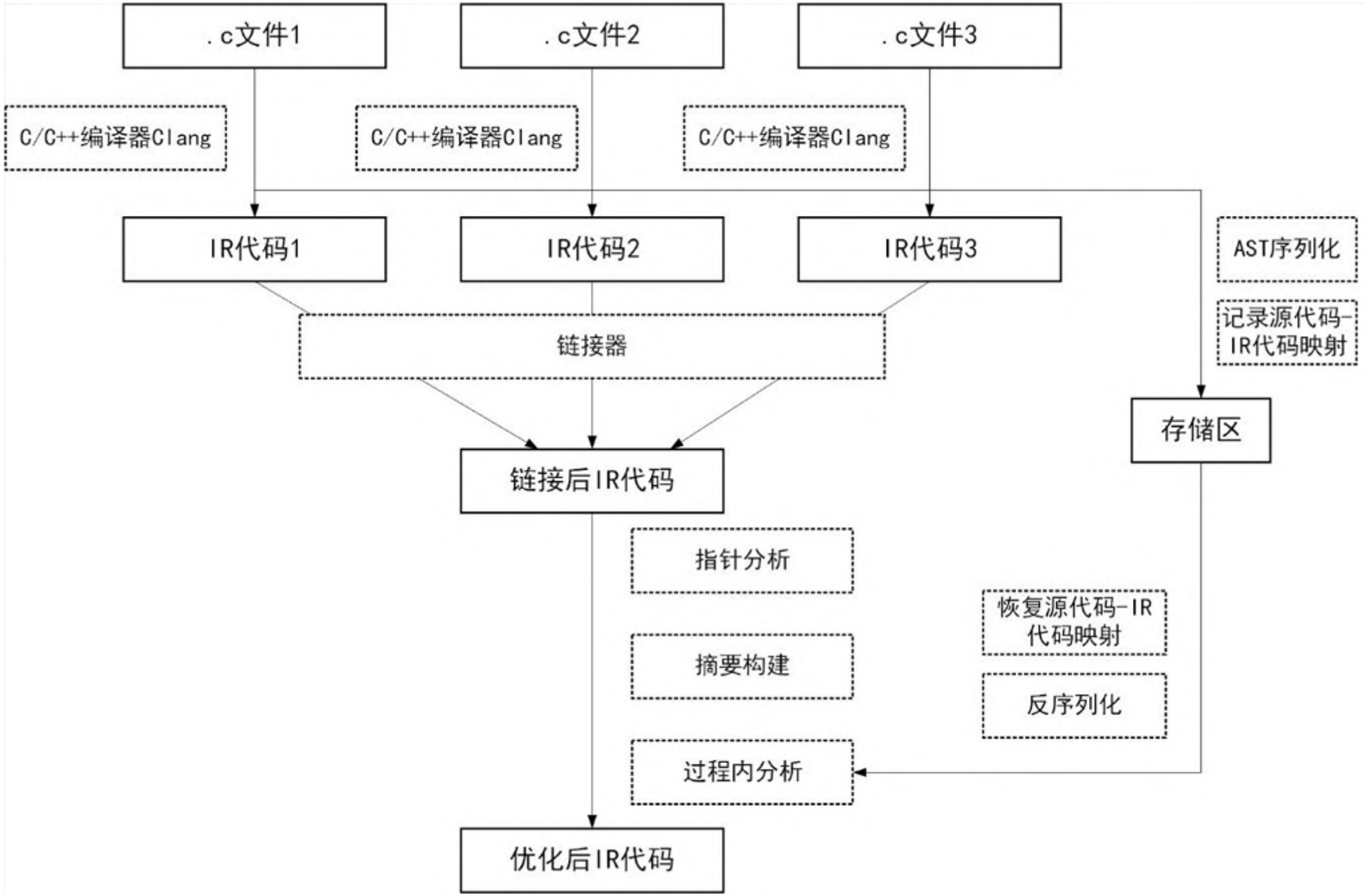


- 绿色：修复位置
- 蓝色：安全位置
- 红色：不安全位置

工具设计

程序崩溃缺陷修复

内存泄漏缺陷修复



实验结果

程序崩溃缺陷修复　内存泄漏缺陷修复

- 实验程序：SPEC2000

Program	Size (Kloc)	#Func	#Allocation
art	1.3	44	11
equake	1.5	45	29
mcf	1.9	44	3
bzip2	4.6	92	10
gzip	7.8	128	5
parser	10.9	342	1
ammp	13.3	197	37
vpr	17.0	290	2
crafty	18.9	127	12
twolf	19.7	209	2
mesa	49.7	1124	67
vortex	52.7	941	8
perlbmk	58.2	1094	4
gap	59.5	872	2
gcc	205.8	2271	53

修复结果

程序崩溃缺陷修复　内存泄漏缺陷修复

- 实验程序：SPEC2000

Program	#Fixed	#Maximum Detected	Percentage(%)	#Fixes	#Useless Fixes
art	0	1	0	0	0
equake	0	0	N/A	0	0
mcf	0	0	N/A	0	0
bzip2	1	1	100	1	0
gzip	1	1	100	1	0
parser	0	0	N/A	0	0
ammp	20	20	100	36	0
vpr	0	0	N/A	0	0
crafty	0	0	N/A	0	0
twolf	0	5	0	0	0
mesa	0	9	0	0	0
vortex	0	0	N/A	0	0
perlbmk	1	8	13	1	0
gap	0	0	N/A	0	0
gcc	2	44	5	2	0
total	25	89	28	41	0

时间开销

- 实验程序：SPEC2000

Program	Compiling and Linking (sec)	LeakFix (sec)		Total (sec)	Percentage(%)
		Pointer Analysis	Fix Analysis		
art	0.20	0.02	0.01	0.23	13.0
equake	0.21	0.01	0.02	0.24	12.5
mcf	1.19	0.02	0.01	1.22	2.5
bzip2	0.36	0.03	0.02	0.41	12.2
gzip	1.31	0.04	0.04	1.39	5.8
parser	1.68	0.18	0.07	1.93	13.0
ammp	2.98	0.12	0.37	3.47	14.1
vpr	2.51	0.20	0.31	3.02	16.9
crafty	3.53	0.16	0.23	3.92	9.9
twolf	6.22	0.27	0.20	6.69	7.0
mesa	9.36	5.36	5.97	20.69	54.8
vortex	9.00	0.94	0.83	10.77	16.4
perlbmk	9.50	18.20	39.20	66.90	85.8
gap	6.03	7.36	22.36	35.75	83.1
gcc	10.99	31.76	95.81	142.99	89.2

内容提要

- 引言
- 基于模式的程序缺陷自动修复框架
- 程序崩溃缺陷的自动修复
- 内存泄漏缺陷的自动修复
- 总结与展望

我们的实践

- 静态代码分析工具CoBOT ，在2015年成为中国首家也是唯一一家通过美国CWE（美国国土安全局资助，国际最权威软件缺陷模式库）认证的软件安全检测类产品，打破静态分析产品国外垄断，漏洞检测水平达到国际先进水平
- 突破了“误报漏报在30%以下的同时程序检测效率达到百万行/小时”这个程序分析领域公认的国际先进水平的检测指标的分析工具，并且目前的检测指标达到“误报漏报在25%以下的同时程序检测效率达到两百万行/小时”

在航空、航天、核能、汽车、核能、金融等30多家单位得到应用

航天领域

北京航天飞控中心

在“探月”项目中，发现6条其他工具未发现缺陷空指针解引用2条、资源泄漏4条

高端制造

高铁通号研究院

在“列控”系统中，发现缓冲区溢出漏洞2条，是其他工具未发现的缺陷

航空领域

中航第一飞机设计院

在“第四代”战机襟缝翼项目中，发现其他工具未发现的2条未初始化使用漏洞

核能

中国广核集团有限公司

在XX系统中，发现内存泄漏4条，是其他工具未发现的缺陷

静态分析技术的未来前景

- 每小时千万行以上的程序静态缺陷扫描技术
  - 程序规模不短增大，持续集成要求分析效率与编译速度同一量级，
- 缺陷模式的全自动总结及挖掘技术
  - 如何利用机器学习技术自动挖掘已知的漏洞模式
- 跨语言分析技术，尤其是对于复杂MVC配置的大型工程
  - 大型系统采用MVC框架，通过配置文件确定数据传输处理的程序
- 动静结合的漏洞自动分析方法的不断完善
  - 采用人工智能技术克服静态分析效率问题以产生更多有效的测试用例


谢谢！