

**课 程 设 计 报 告**

**题目： 基于源代码的软件同源性分析**

**与漏洞检测系统**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **项目** | **撰写规范** | **课程设计过程** | **问题分析与小结** | **总分** | **教师签字** |
| 分值 | 20 | 50 | 30 | 100 |  |
| 评分 |  |  |  |  |  |

**课程名称： 软件安全课程设计**

**专业班级： 信息安全1602班**

**学 号： U201614848**

**姓 名： 郭倜维**

**指导教师： 付才**

**报告日期： 2018年10月25日**

**计算机科学与技术学院**

|  |  |  |
| --- | --- | --- |
| **序号** | **内容** | **要求** |
| A1 | 利用字符串匹配进行同源性检测 | 通过代码有效字符串对比匹配，分析样本之间的拷贝比率。 |
| A2 | 利用控制流程图CFG进行源代码同源性检测 | 通过提取代码的调用关系图，检测样本之间各个函数调用关系图是否相似，得出相似的概率 |
| A3 | 跨语言同源性检测验证 | 在软件版权保护中，有时候需要检测是否参考了有版权的代码，换用一种语言实现同样的功能，本功能可以通过 CFG 检测实现，但需要给出 4 个以上不用语言的同源性分析样本 |
| B1 | 栈缓冲区检测 | 根据栈缓冲区原理分析分配的栈数据区是否存在溢出的问题，给出可疑代码行数与列数。 |
| B2 | 堆缓冲区检测 | 根据堆缓冲区原理分析分配的栈数据区是否存在溢出的问题，给出可疑代码行数与列数。 |
| B3 | 整数宽度溢出检测 | 根据整数宽度溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数 |
| B4 | 整数运算溢出检测 | 根据整数运算溢出原理分析分配的数据是否存在溢出的问题，给出可疑代码行数与列数。 |
| B5 | 整数符号溢出检测 |  |
| B6 | 格式化字符串漏洞检测 |  |
| B7 | 竞争性条件 |  |
| B8 | 空指针引用 |  |
| C1 | 支持分布式任务调度 | 需要设计一个主控，多个进程/主机并发检测。 |
| C2 | 提供系统界面 | 上述所有功能要有图形界面展示 |
| C3 | 同源性检测样本库 | 样本数大于等于 50 个，每个代码行数不 少于100行,包含1-100行相同代码。 |
| C4 | 漏洞检测样本库 | 样本数大于等于 50 个，每个代码行数不 少于100行；每种漏洞至少一个。 |

**目录**

[1 绪言 6](#_Toc528314855)

[1.1 课程设计背景 6](#_Toc528314856)

[1.2 课程设计的目的与意义 6](#_Toc528314857)

[1.3 课程设计的主要内容 6](#_Toc528314858)

[2 设计思路 7](#_Toc528314859)

[2.1 编程思路 7](#_Toc528314860)

[2.1.1 基于字符串比对的同源性检测 7](#_Toc528314861)

[2.1.2 基于CFG同源性检测 7](#_Toc528314862)

[2.1.3 栈溢出漏洞检测 8](#_Toc528314863)

[2.1.4 堆溢出漏洞检测 8](#_Toc528314864)

[2.1.5 整数宽度溢出漏洞检测 8](#_Toc528314865)

[2.1.6 整数运算溢出漏洞检测 8](#_Toc528314866)

[2.2 算法设计 9](#_Toc528314867)

[2.2.1 文件预处理算法 9](#_Toc528314868)

[2.2.2 字符串匹配度计算算法 14](#_Toc528314869)

[2.2.3 函数体判断函数 16](#_Toc528314870)

[2.2.4 敏感函数判断算法 17](#_Toc528314871)

[2.2.5 函数参数分析算法 18](#_Toc528314872)

[2.2.6 函数体链表构造算法 19](#_Toc528314873)

[2.2.7 CFG图比较算法 24](#_Toc528314874)

[2.2.8 普通变量判断和分析算法 26](#_Toc528314875)

[2.2.9 敏感函数分析算法 29](#_Toc528314876)

[2.2.10 缓冲区变量分析算法 31](#_Toc528314877)

[2.2.11 赋值语句判断和分析算法 32](#_Toc528314878)

[2.2.12 栈溢出检测算法 34](#_Toc528314879)

[2.2.13 堆溢出检测算法 37](#_Toc528314880)

[2.2.14 整数宽度溢出检测算法 40](#_Toc528314881)

[2.2.15 整数计算溢出检测算法 42](#_Toc528314882)

[2.3 数据结构设计 44](#_Toc528314883)

[2.3.1 函数体结构体 44](#_Toc528314884)

[2.3.2 函数结构体 46](#_Toc528314885)

[2.3.3 敏感函数结构体 47](#_Toc528314886)

[2.3.4普通变量结构体 48](#_Toc528314887)

[2.3.5 缓冲区变量结构体 48](#_Toc528314888)

[2.3.6 边界检查语句结构体 49](#_Toc528314889)

[2.3.7 赋值语句结构体 49](#_Toc528314890)

[2.3.8 运算语句结构体 50](#_Toc528314891)

[2.4 样本库设计 50](#_Toc528314892)

[2.4.1 同源性检测样本库 50](#_Toc528314893)

[2.4.2 漏洞检测样本库 51](#_Toc528314894)

[2.5 关键问题及其解决方法 51](#_Toc528314895)

[2.5.1 漏洞检测原理 51](#_Toc528314896)

[2.5.2 CFG的存储问题 51](#_Toc528314897)

[3 程序设计 52](#_Toc528314898)

[3.1 系统各功能程序设计 52](#_Toc528314899)

[3.1.1利用字符串匹配进行同源性检测 52](#_Toc528314900)

[3.1.2利用CFG图进行同源性检测 52](#_Toc528314901)

[3.1.3 栈溢出漏洞检测 54](#_Toc528314902)

[3.1.4 堆溢出漏洞检测 55](#_Toc528314903)

[3.1.5 整数宽度溢出漏洞检测 57](#_Toc528314904)

[3.1.6 整数运算溢出漏洞检测 59](#_Toc528314905)

[4 系统测试 61](#_Toc528314906)

[4.1利用字符串匹配进行同源性检测 61](#_Toc528314907)

[4.2利用CFG图进行同源性检测 62](#_Toc528314908)

[4.3栈溢出漏洞检测 63](#_Toc528314909)

[4.4堆溢出漏洞检测 64](#_Toc528314910)

[4.5整数宽度溢出漏洞检测 64](#_Toc528314911)

[4.6整数运算溢出漏洞检测 65](#_Toc528314912)

[5 总结与体会 65](#_Toc528314913)

# 1 绪言

## 课程设计背景

计算机软件在全球经济信息一体化进程中扮演的角色越来越重要，而以软件为核心的软件产业目前已经成为21世纪信息产业的重要组成部分。计算机软件的技术含量高，且具有巨大的潜在经济价值，使得自20世纪60年代软件产业兴起开始，出现了计算机软件被盗用、被侵权的现象，并且随着软件产业的发展愈演愈烈。这个时候，“软件安全”就应运而生了。

软件安全，就是我们平时说的使软件在受到恶意攻击的情形下依然能够继续正确运行及确保软件被在授权范围内合法使用的思想，同时还需包括保护软件中的智力成果、知识产权不被非法使用，包括篡改及盗用等。

## 课程设计的目的与意义

在完成了《C语言》和《软件安全》等专业课程基础上，本次课程设计中设计了同源性检测和漏洞检查两个部分；软件同源性检测技术能有效识别软件之间的相似程度，为软件知识产权保护提供了有力的证据；漏洞扫描技术可以事先发现可能产生漏洞的代码，为软件开发人员对软件的维护帮助极大。

学生可以利用所学知识完成代码的同源性检测和漏洞检测，在这个过程中，学生一方面可以模拟相关软件的工作模式，培养思考问题和解决问题的能力，另一方面可以巩固所学知识，提高自己的代码的安全性和健壮性。从而通过对本次课设的实现能够加深学生对于软件安全知识及相关领域的理解。

## 课程设计的主要内容

1. 利用字符串匹配进行同源性检测；
2. 利用控制流程图CFG进行源代码同源性检测；
3. 栈缓冲区溢出检测；
4. 堆缓冲区溢出检测；
5. 整数宽度溢出检测；
6. 整数运算溢出检测；
7. 支持分布式任务调度；
8. 实现同源性检测样本库与漏洞检测样本库；
9. 提供系统界面；

# 设计思路

## 编程思路

### 基于字符串比对的同源性检测

1. 打开两个文件；
2. 对两个文件进行预处理；
3. 将处理后的文件内容分别重新写入一个新的文件；
4. 对预处理产生的两个文件进行逐行读取进行字符串比较；
5. 计算两个文件的字符串相似度；
6. 计算两个文件代码的相似度。

### 基于CFG同源性检测

1. 打开两个文件；
2. 对两个文件进行预处理；
3. 将处理后的文件内容分别重新写入一个新的文件；
4. 利用预处理产生的两个文件构造代码中的函数调用关系图；
5. 计算两个关系图的相似度；
6. 计算两个文件代码的相似度。

### 栈溢出漏洞检测

1. 分析栈溢出发生的原理；
2. 构造代码中与栈溢出有关的变量构成的链表；
3. 构造代码中与栈溢出有关的函数构成的链表；
4. 遍历代码，分析代码中与栈溢出有关的操作，通过分类讨论判断是否会发生栈溢出，获取栈溢出可能代码所在位置和该代码。

### 堆溢出漏洞检测

1. 分析堆溢出发生的原理；
2. 构造代码中与堆溢出有关的变量构成的链表；
3. 构造代码中与堆溢出有关的函数构成的链表；
4. 构造代码中与堆溢出有关的操作如边界检查等的链表；
5. 遍历代码，分析代码中与堆溢出有关的操作，通过分类讨论判断是否会发生堆溢出，获取堆溢出可能代码所在位置和该代码。

### 整数宽度溢出漏洞检测

1. 分析整数宽度溢出发生的原理
2. 构造代码中整数变量的链表；
3. 构造代码中可能发生溢出的操作的链表；
4. 遍历代码，分析代码中与溢出有关的操作，通过分类讨论判断是否会发生溢出，获取溢出可能代码所在位置和该代码。

### 整数运算溢出漏洞检测

1. 分析整数运算溢出发生的原理；
2. 构造代码中整数变量的链表；
3. 构造代码中可能发生溢出的操作的链表；
4. 遍历代码，分析代码中与溢出有关的操作，通过分类讨论判断是否会发生溢出，获取溢出可能代码所在位置和该代码。

## 算法设计

### 文件预处理算法

对于文件a.cpp，里面存储的是一套代码，为了提高之后同源性检测和溢出检测的准确率和准确度，需要对文件中的代码进行一定程度的预处理，也就是去除C语言标准注释、C++标准注释、预编译语句、空行和行前空格。

算法实现的是打开名为“In\_name”的文件，对其中的代码进行处理，然后将处理后的代码放入名为“Out\_name”的文件中。

算法实现过程中采用逐行的读取文件方式，然后对读取出来的内容进行是否为注释、空格以及预编译语句等的判断，然后根据判断结果进行处理。对于注释符号的判断可以采用标志符的方式确定，例如在第二种算法中，采用了两个标志符，分别是左标志符和右标志符，在左标志符满足要求的情况下判断右标志符号是否满足要求就可以确定是不是注释了。

逐行获取文件内容有两种实现方式，一种是getline函数，另一种是fgets函数；与之相对应，采用不同的方式打开文件，前者使用文件流的方式，后者使用fopen函数打开文件；根据不同函数的使用方法，可以设计出不同的文件预处理算法；具体算法实现如下。

算法void clean\_1(string name)实现如下：

void clean\_1(string name)

{

ifstream infile(name.c\_str()); //读取输入文件

ofstream outfile1("1\_c.cpp"); //把消除注释后的程序保存到"无注释.cpp"中

unsigned long i, j, size; //整型初始值

string instring,outstring; ////字符串 instring,outstring

bool flag; //标记

while(getline(infile,instring)) //读取文件内容

{

outstring=""; //outstring为空格

size=instring.size(); //字符串大小

flag=false; //flag为false

for(i=0;i<size;) //循环语句

{

switch(instring[i])//swich语句

{

case '/':

if(instring[i+1]=='/') //如果找到“//”

flag=true; //flag为true

else if(instring[i+1]=='\*') //如果找到“/\*\*/”中的 “/\*”

{

i=instring.find("\*/",i+2); //查找“\*/”的位置，用i表示

while(i==-1) //找到当的不是“\*/”时

{

if(!getline(infile,instring))//读取文件内容

{

//cout<<"not matching"<<endl; //打印出"not matching"

return ; //返回0

}

size=instring.size(); //继续查找

flag=false; //flag为false

i=instring.find("\*/"); //查找“\*/”的位置，用i表示

}

i+=2; //i=i+2

}

else //否则

outstring+=instring[i++]; //把非注释字符存放到outstring

break; //跳出

case '"':

for(j=i+1;j<size;++j)

{

if(instring[j]=='"'&&instring[j-1]!='\\') //如果查找“"”

break; //跳出

}

if(j==size) //若找不到

{

//cout<< "not matching"<<endl; //输出"not matching"

return ; //返回0

}

outstring+=instring.substr(i,j-i+1); //把非注释字符保存到outstring中

i=j+1;

break;

default: outstring+=instring[i++]; //其他，将非注释字符存放到outstring中

}

if(flag) break; //如果标记为0，跳出

}

//cout <<outstring<<endl; //输出outstring输出

outfile1<<outstring<<endl; //把字符串outstring保存到outfile1

}

ifstream fin("1\_c.cpp"); //读取"无注释.cpp"

anay\_code\_1("1\_c.cpp"); //调用int anay\_code(char \*path)

return ; //返回0

}

算法Pre\_Func\_deal(FILE\* F\_In,FILE\* F\_Out,char In\_name[30],char Out\_name[20]实现如下：

void Pre\_Func\_deal(FILE\* F\_In,FILE\* F\_Out,char In\_name[30],char Out\_name[20])

{

char line[MAX];

char line\_temp[MAX];

char \*pc;

char \*Left\_flag,\*Right\_flag; //注释符检测左标志与右标志

F\_In=fopen(In\_name,"r"); //打开输入文件

F\_Out=fopen(Out\_name,"w"); //打开输出文件

while(!feof(F\_In)) //判断文件尾

{

while(fgets(line,MAX,F\_In)) //逐行读取文件 遇到换行符结束

{

Left\_flag=strstr(line,"/\*"); //是否存在 /\*

if(Left\_flag) //找到左标志

{

\*Left\_flag='\n'; //去掉注释内容

\*(Left\_flag+1)='\0';

strcpy(line\_temp,line);

Right\_flag=strstr(Left\_flag+2,"\*/"); //是否存在 \*/

while(Right\_flag == NULL) //该行不存在右标志，继续读取下一行，直到找到为止

{

fgets(line,MAX,F\_In);

Right\_flag=strstr(line,"\*/");

}

strcpy(line,line\_temp);

}

pc=strstr(line,"#"); //预编译语句判断

if(pc)

{

\*pc='\n'; //去掉预编译语句

\*(pc+1)='\0';

}

pc=strstr(line,"//"); //注释符判断

if(pc)

{

\*pc='\n'; //去掉注释内容

\*(pc+1)='\0';

}

pc=line;

while(isspace(\*pc)) //去掉字符串开头的空格

pc++;

fputs(pc,F\_Out); //逐行写入文件

}

}

fclose(F\_In);

fclose(F\_Out);

}

### 字符串匹配度计算算法

对文件1.cpp和2.cpp进行字符串匹配度计算，只需要从文件1.cpp逐行读取字符串，然后对每次读取从1.cpp读取的字符串与2.cpp文件中读取每一行进行比较，若找到相同的则在匹配度计数器上加一，最后计算1.cpp文件中与2.cpp文件中相同的与1.cpp文件中字符串总数量的比值即为所求的匹配度；算法的具体实现如下。

字符串匹配度计算算法实现如下：

if(in\_1) // 有该文件

{

clean\_2("2.cpp");

while (getline (in\_1, line\_1[i])) // line中不包括每行的换行符

{

string::iterator it = line\_1[i].begin();

while ((it=find(it, line\_1[i].end(), ' '))!=line\_1[i].end())

{

line\_1[i].erase(it);

if (it == line\_1[i].end())

break;

}

if(line\_1[i].length() != 0)

{

//line\_1 = str.replaceAll(regex, "");

if(line\_1[i] != "\\s+")

{

//cout << line\_1[i] << endl;

ifstream in\_2("2\_c.cpp");

string line\_2[1000];

if(in\_2) // 有该文件\*/

{

while (getline (in\_2, line\_2[j])) // line中不包括每行的换行符

{

string::iterator it2 = line\_2[j].begin();

while ((it2=find(it2, line\_2[j].end(), ' '))!=line\_1[j].end())

{

line\_2[j].erase(it2);

if (it2 == line\_2[j].end())

break;

}

if(line\_2[j].length() != 0)

{

if(line\_2[j] != "\\s+")

{

if(line\_1[i]==line\_2[j])

{

k++;

break;

}

}

}

//cout << line\_2[j] << endl;

j++;

}

j = 0;

}

else // 没有该文件

{

cout <<"no such file" << endl;

break;

}

}

i++;

}

}

}

else // 没有该文件

{

cout <<"no such file" << endl;

}

printf("\nk = %d\ni = %d\n",k, i);

printf("rate = %.10f\n",(double)k/(double)i);

### 函数体判断函数

根据编程思路，分析代码的时候需要将代码中的函数体构造的的结构体链接起来生成一个函数体链表，通过遍历这个链表就可以获得代码中函数体的信息，从而可以通过这个链表实现依次对每个函数体同源性和溢出的判断。而函数体链表构造需要先获取函数体的位置，也就是判断读取的语句是否为函数体的开头；利用函数strstr（）可以判断字符串是否存在，从而就实现了函数体的判断；具体算法实现如下。

算法bool Judge\_body(char str[MAX])实现如下：

bool Judge\_body(char str[MAX])

{

bool flag;

if(strstr(str,"(")!=NULL)

{

if(strstr(str,"char")!=NULL || strstr(str,"int")!=NULL || strstr(str,"long")!=NULL || strstr(str,"float")!=NULL || strstr(str,"double")!=NULL || strstr(str,"void")!=NULL)

{

if(strstr(str,";")==NULL)

flag = true;

else

flag = false;

}

else

flag = false;

}

else

flag = false;

return flag;

}

### 敏感函数判断算法

对于敏感函数的判断，只需要判断读取的语句中是否存在敏感函数的函数名就可以了，同样采用strstr（）函数，判断算法的具体实现如下。

算法bool Judge\_func\_m(char str[MAX])实现如下：

bool Judge\_func\_m(char str[MAX])

{

bool flag;

if(strstr(str,"strcpy")!=NULL || strstr(str,"strncpy")!=NULL || strstr(str,"memcpy")!=NULL ||strstr(str,"strcat")!=NULL || strstr(str,"strncpy")!=NULL || strstr(str,"sscacnf")!=NULL)

flag = true;

else

flag = false;

return flag;

}、

### 函数参数分析算法

对于函数结构体，需要获取该函数的参数对其进行赋值，函数参数的获取同样使用函数strstr()获取函数的返回值、函数的名字以及函数的参数类型。

void Func\_Analyze(char str[MAX],struct Func\* func)

{

int c1=0;

sscanf(str,"%[^ ]",func->back\_value);

str=strstr(str," ");

str++;

sscanf(str,"%[^(]",func->func\_name);

str=strstr(str,"(");

str++;

c1=Count\_Char(str,",");

func->para\_num=c1+1;

switch(c1)

{

case 0:

sscanf(str,"%[^ ]",func->para\_type\_1);

break;

case 1:

sscanf(str,"%[^ ]",func->para\_type\_1);

str=strstr(str,",");

str++;

sscanf(str,"%[^ ]",func->para\_type\_2);

break;

case 2:

sscanf(str,"%[^ ]",func->para\_type\_1);

str=strstr(str,",");

str++;

sscanf(str,"%[^ ]",func->para\_type\_2);

str=strstr(str,",");

str++;

sscanf(str,"%[^ ]",func->para\_type\_3);

break;

default:

sscanf(str,"%[^ ]",func->para\_type\_1);

str=strstr(str,",");

str++;

sscanf(str,"%[^ ]",func->para\_type\_2);

str=strstr(str,",");

str++;

sscanf(str,"%[^ ]",func->para\_type\_3);

str=strstr(str,",");

str++;

sscanf(str,"%[^)]",func->para\_remain);

break;

}

}

### 函数体链表构造算法

对于函数体链表的构造，首先需要判断读取的内容是否为函数体的内容，在确定是函数体后，利用多重判断语句获取函数体结构体中各个元素的值，也就是说需要对该函数体中子函数、变量、运算语句以及赋值语句等构成的链表进行初始化和生成。对于是否在同一个函数体中，采用两个标志符来确定，分别是brace\_l和brace\_r，brace\_l表示读到的左大括弧的数量，brace\_r表示读到的右大括弧的数量，两者的差为0时表示读取完了一个函数体。函数体节点中的指针都是指向链表的头节点，里面没有存信息；函数体链表构造算法具体实现如下。

算法struct Func\* funcchain(FILE \*pf)实现如下：

struct Func\* funcchain(FILE \*pf)

{

int row=0;

int brace\_l=0, brace\_r=0, brace=0;

char buf\_1[MAX]={0};

char buf\_2[MAX]={0};

struct Func \*func\_head,\*func\_tail,\*func\_new;

struct Variate \*v\_head,\*v\_tail,\*v\_new;

struct Func\_m \*m\_head,\*m\_tail,\*m\_new;

struct Buffer \*buf\_head,\*buf\_tail,\*buf\_new;

struct Operation \*op\_head,\*op\_tail,\*op\_new;

func\_head = (struct Func\*)malloc(sizeof(struct Func));

func\_tail = func\_head;

func\_tail->func\_next = NULL;

while(!feof(pf))

{

fgets(buf\_1, MAX, pf);

row++;

if(Judge\_body(buf\_1))

{

func\_new = (struct Func\*)malloc(sizeof(struct Func));

func\_new->start\_row = row;

buf\_head = (struct Buffer\*)malloc(sizeof(struct Buffer));

buf\_tail = buf\_head;

buf\_tail->buf\_next = NULL;

v\_head = (struct Variate\*)malloc(sizeof(struct Variate));

v\_tail = v\_head;

v\_tail->v\_next = NULL;

m\_head = (struct Func\_m\*)malloc(sizeof(struct Func\_m));

m\_tail = m\_head;

m\_tail->m\_next = NULL;

op\_head = (struct Operation\*)malloc(sizeof(struct Operation));

op\_tail = op\_head;

op\_tail->o\_next = NULL;

func\_new->va\_chain = v\_head;

func\_new->mf\_chain = m\_head;

func\_new->buf\_chain = buf\_head;

func\_new->op\_chain = op\_head;

func\_tail->func\_next=func\_new;

func\_tail=func\_new;

func\_tail->func\_next=NULL;

brace\_l = Count\_char(buf\_1,"{");

brace = brace + brace\_l;

brace\_r = Count\_char(buf\_1,"}");

brace = brace - brace\_r;

do

{

fgets(buf\_2,MAX,pf);

row++;

brace\_l = Count\_char(buf\_2,"{");

brace = brace + brace\_l;

brace\_r = Count\_char(buf\_2,"}");

brace = brace - brace\_r;

if(strstr(buf\_2,"char")!=NULL || strstr(buf\_2,"short")!=NULL || strstr(buf\_2,"int")!=NULL || strstr(buf\_2,"long")!=NULL)

{

if(strstr(buf\_2,"(")==NULL)

{

if(strstr(buf\_2,"\*")!=NULL || strstr(buf\_2,"[")!=NULL)

{

buf\_new = (struct Buffer\*)malloc(sizeof(struct Buffer));

Buffer\_analyze(buf\_2, buf\_new);

buf\_tail->buf\_next = buf\_new;

buf\_tail = buf\_new;

buf\_tail->buf\_next = NULL;

}

else

{

v\_new = (struct Variate\*)malloc(sizeof(struct Variate));

Variate\_analyze(buf\_2, v\_new);

v\_tail->v\_next = v\_new;

v\_tail = v\_new;

v\_tail->v\_next = NULL;

}

}

}

if(Judge\_func\_m(buf\_2))

{

m\_new = (struct Func\_m\*)malloc(sizeof(struct Func\_m));

m\_new->row = row;

Func\_m\_analyze(buf\_2, m\_new);

m\_tail->m\_next = m\_new;

m\_tail = m\_new;

m\_tail->m\_next = NULL;

}

if(Judge\_operation(buf\_2)) //扫描有无运算语句

{

op\_new=(struct Operation\*)malloc(sizeof(struct Operation));

op\_new->row=row;

Operation\_analyze(buf\_2, op\_new);

op\_tail->o\_next=op\_new;

op\_tail=op\_new;

op\_tail->o\_next=NULL;

}

}

while(brace);

func\_new->end\_row = row;

}

}

return func\_head;

}

### CFG图比较算法

利用CFG图进行同源性检测，关键在于生成一个CFG图。在生成一个CFG图之后，依次对每个函数体进行同源性检测，直到最后一个函数体检测结束。在函数体内统计函数名相同的子函数的个数n和函数体内全部子函数的个数N，计算其比例作为该函数体同源性比例，然后求所有函数体的平均值为最后的结果。

tree\_2\_head = tree\_2->next;

while(tree\_1->next != NULL)

{

flag\_1 = 0; //

tree\_1 = tree\_1->next;

functree\_1 = tree\_1->head; //functree\_1指向tree\_1指向的函数链表

functree\_1 = functree\_1->next;

c2++;

strcpy(tree\_name\_1, functree\_1->func\_name); //获得函数体链表1的函数体名字

while(tree\_2->next != NULL)

{

tree\_2 = tree\_2->next;

functree\_2 = tree\_2->head;

functree\_2 = functree\_2->next; //指向函数体函数链表函数体名字

strcpy(tree\_name\_2, functree\_2->func\_name);

if(!strcmp(tree\_name\_1, tree\_name\_2)) //比较函数体名字是否相同

{

flag\_1 = 1;

functree\_2\_head = functree\_2;

while(functree\_1->next != NULL)

{

flag\_2 = 1;

functree\_2 = functree\_2\_head; //指向函数体2的函数链表首节点

functree\_1 = functree\_1->next; //函数体1的函数链表中的函数

strcpy(func\_name\_1, functree\_1->func\_name);

i++; //函数体1中的函数个数

while(functree\_2->next != NULL)

{

functree\_2 = functree\_2->next;

strcpy(func\_name\_2, functree\_2->func\_name);

if(!strcmp(func\_name\_1,func\_name\_2))

c1++; //函数体1和2中函数名相同的个数

}

}

if(flag\_2)

{

rate\_line = c1\*100/i;

rate\_chain[j] = rate\_line;

}

else

{

rate\_chain[j] = 50;

}

j++;

i = 0;

c1 = 0;

flag\_2 = 0;

}

}

tree\_2 = tree\_2\_head;

if(!flag\_1) //不存与在函数体1相同的函数体2

{

rate\_chain[j] = 0.0;

j++;

}

}

for(i=0;i<20;i++)

{

rate=rate\_chain[i]+rate;

}

rate=rate/c2; //函数体链表中函数体个数c2

printf("函数调用关系的相似度至少为：%.3f(百分制)",rate);

### 普通变量判断和分析算法

扫描函数体后，通过多重判断可以获取函数体中使用到的变量，但是在溢出检测过程中还需要获取变量的使用信息，根据编程思路中，我们会构造一条函数体中出现的普通变量构成的链表，链表是由若干个普通变量的结构体节点链接出来的，对于每个节点都存储着这个变量的信息，而分析算法就是对这个变量对应的节点的各个值进行赋值。同样利用strstr()函数、sscanf()函数和正则表达式获取各个部分的内容。对于变量的空间大小size，对于无符号整数和有符号整数需要进一步分析；具体算法实现如下。

算法void Variate\_analyze(char str[MAX],struct Variate\* var)实现如下：

void Variate\_analyze(char str[MAX],struct Variate\* var)

{

char temp[10]={0};

if(strstr(str,"short ")!=NULL ||strstr(str,"int ")!=NULL || strstr(str,"long ")!=NULL)

{

if(strstr(str,"unsigned ")!=NULL)

{

sscanf(str,"%[^ ]",var->type);

strcat(var->type," ");

str=strstr(str," ");

str++;

sscanf(str,"%[^ ]",temp);

strcat(var->type,temp);

str=strstr(str," ");

str++;

sscanf(str,"%[^;]",var->name);

}

else

{

sscanf(str,"%[^ ]",var->type);

str=strstr(str," ");

str++;

sscanf(str,"%[^;]",var->name);

}

if(strstr(var->type,"short")!=NULL)

{

if(strstr(var->type,"unsigned ")!=NULL)

{

var->type\_flag=2;

}

else

{

var->type\_flag=1;

}

}

if(strstr(var->type,"int")!=NULL)

{

if(strstr(var->type,"unsigned ")!=NULL)

{

var->type\_flag=4;

}

else

{

var->type\_flag=3;

}

}

if(strstr(var->type,"long")!=NULL)

{

if(strstr(var->type,"unsigned ")!=NULL)

{

var->type\_flag=6;

}

else

{

var->type\_flag=5;

}

}

}

}

### 敏感函数分析算法

敏感函数的分析是一个比较复杂的过程，首先需要判断该敏感函数是哪种敏感函数，在确定敏感函数类型后根据不同类型的敏感函数的参数不同位置获取相应的参数，并将获取的参数存入到该敏感函数结构体中；具体算法如下所示。

算法void Func\_m\_analyze(char str[MAX],struct Func\_m\* m\_func)实现如下：

void Func\_m\_analyze(char str[MAX],struct Func\_m\* m\_func)

{

char temp[20] = {0};

if(strstr(str,"strcpy")!=NULL || strstr(str,"strncpy")!=NULL || strstr(str,"memcpy")!=NULL ||strstr(str,"strcat")!=NULL || strstr(str,"sscanf")!=NULL)

{

sscanf(str,"%[^(]",m\_func->name);

if(!strcmp(m\_func->name,"strcpy") || !strcmp(m\_func->name,"strcat"))

{

str=strstr(str,"(");

str++;

sscanf(str,"%[^,]",m\_func->des); //目的参数

str=strstr(str,",");

str++;

sscanf(str,"%[^)]",m\_func->sou); //源参数

}

else

{

if(!strcmp(m\_func->name,"strncpy") || !strcmp(m\_func->name,"memcpy"))

{

str=strstr(str,"(");

str++;

sscanf(str,"%[^,]",m\_func->des);

str=strstr(str,",");

str++;

sscanf(str,"%[^,]",m\_func->sou);

str=strstr(str,",");

str++;

sscanf(str,"%[^)]",temp);

m\_func->n=atoi(temp);

}

else

{

if(!strcmp(m\_func->name,"sscanf"))

{

str=strstr(str,"(");

str++;

sscanf(str,"%[^,]",m\_func->sou);

str=strstr(str,",");

str++;

str=strstr(str,",");

str++;

sscanf(str,"%[^)]",m\_func->des);

}

}

}

}

}

### 缓冲区变量分析算法

缓冲区变量主要分为两个部分，一个是数组，一个是malloc分配的缓冲区变量；所以对不同的缓冲区变量需要进行不同的处理才能得到该变量的参数等。对于数组，同样可能存在两种情况，一种是直接定义了数组大小的数组，另一种是通过对数组进行初始化间接定义数组大小的数组；对这两种不同的数组照样进行判断，区分后进行分析。对于malloc分配的缓冲区变量，可以直接通过分配内存的数据类型得到该缓冲区变量的类型；算法的具体实现如下：

算法void Buffer\_analyze(char str[MAX],struct Buffer\* buf)实现如下：

void Buffer\_analyze(char str[MAX],struct Buffer\* buf)

{

char temp[MAX]={0};

if(strstr(str,"[")!=NULL) //数组变量

{

if(strstr(str,"char ")!=NULL || strstr(str,"int ")!=NULL || strstr(str,"long ")!=NULL)

{

sscanf(str,"%[^ ]",buf->type);

str=strstr(str," ");

str++;

sscanf(str,"%[^[]",buf->name);

str=strstr(str,"[");

str++;

}

if(\*str!=']') //直接给出数组定义

{

sscanf(str,"%[^]]",temp);

buf->size=atoi(temp);

}

else //初始化数组

{

str=strstr(str,"=");

sscanf(str,"%[^;]",temp);

buf->size=strlen(temp)-2;

}

}

if(strstr(str,"char\* ")!=NULL || strstr(str,"short\* ")!=NULL || strstr(str,"int\* ") || strstr(str,"long\* ")!=NULL)

{

if(strstr(str,"malloc")!=NULL)

{

sscanf(str,"%[^\*]",buf->type);

str=strstr(str,"\* ");

str++;str++;

sscanf(str,"%[^=]",buf->name);

}

else

{

sscanf(str,"%[^\*]",buf->type);

str=strstr(str,"\* ");

str++;str++;

sscanf(str,"%[^;]",buf->name);

}

}

}

### 赋值语句判断和分析算法

对读取的语句是否为赋值语句的算法为bool Judge\_equal(char str[MAX])；

对赋值语句的分析算法为struct Equal\* Equal\_analyze(char str[MAX],struct Equal\* equal)。

算法bool Judge\_equal(char str[MAX])实现如下：

bool Judge\_equal(char str[MAX])

{

char\* temp\_1;

char temp\_2[20] = {0};

int i;

bool flag\_1 = true;

bool flag = true;

temp\_1=str;

/\*if(strstr(str,"=")!=NULL && strstr(str,">=")==NULL && strstr(str,"<=")==NULL && strstr(str,"==")==NULL && strstr(str,"!=")!=NULL)

{

flag = true;

}

else

flag = false;\*/

if(strstr(str,"=")!=NULL)

{

temp\_1 = strstr(temp\_1,"=");

temp\_1++;

sscanf(temp\_1,"%[^;]",temp\_2);

for(i=0;i<strlen(temp\_2);i++)

{

if(!(temp\_2[i]>='0' && temp\_2[i]<='9'))

continue;

else

{

flag = false;

break;

}

}

if(flag\_1 == false)

flag = false;

}

else

flag = false;

return flag;

}

算法struct Equal\* Equal\_analyze(char str[MAX],struct Equal\* equal)如下：

struct Equal\* Equal\_analyze(char str[MAX],struct Equal\* equal)

{

sscanf(str, "%[^=]", equal->left);

str = strstr(str, "=");

str++;

sscanf(str, "%[^;]", equal->right);

return equal;

}

### 栈溢出检测算法

分析栈溢出发生原理，根据具有栈溢出发生可能性的几种情况设计栈溢出判断语句，在栈溢出检测过程中，可以使用一个标志符over\_flow来进行标记发生栈溢出可能性的地方；具体算法的实现如下：

while (m->m\_next != NULL)

{

flag = 1;

m = m->m\_next;

//取出敏感函数的实参

strcpy(str\_des,m->des);

strcpy(str\_sou,m->sou);

n = m->n;

while(v->v\_next != NULL)

{

v=v->v\_next;

//获取敏感函数使用时变量的大小

if(!strcmp(str\_des,v->name))

v\_size\_des=v->size;

if(!strcmp(str\_sou,v->name))

v\_size\_sou=v->size;

}

//对不同敏感函数的溢出进行判断

if(!strcmp(m->name,"strcpy") || !strcmp(m->name,"sscanf"))

{

if(v\_size\_des<v\_size\_sou)

over\_flow=1; //明显大于缓冲空间的，一定会溢出

else

over\_flow=3; //没有明显大于，不会溢出

}

else

{

if(!strcmp(m->name,"strncpy") || !strcmp(m->name,"memcpy"))

{

if(v\_size\_des<n)

over\_flow=1;

else

over\_flow=3;

}

else

{

if(!strcmp(m->name,"strcat"))

{

if(v\_size\_des<(v\_size\_des+v\_size\_sou))

over\_flow=1;

else

over\_flow=3;

}

}

}

if(over\_flow==1)

{

printf("Overflow in line %d:\n",m->row);

Show(pf2,m->row,stroutput);//打印溢出函数

}

}

if(flag==0)

printf("No overflow:No such a function can cause overflow!\n"); //无敏感函数

else

if(over\_flow==3)

printf("No overflow:Function in line %d cannot cause overflow!\n",m->row); //有敏感函数但不会产生溢出

### 堆溢出检测算法

分析堆溢出发生原理，根据具有堆溢出发生可能性的几种情况设计堆溢出判断语句，在堆溢出检测过程中，可以使用两个标志符flag\_op和flag\_m来进行标记不同情况，然后利用两个标志符的值最后确定发生堆溢出可能性的地方；具体算法的实现如下：

while(m->m\_next != NULL)

{

flag\_m = 1;

m = m->m\_next;

//取出敏感函数的实参

strcpy(str\_des,m->des);

strcpy(str\_sou,m->sou);

n = m->n;

while(buf->buf\_next != NULL)

{

buf = buf->buf\_next;

if(!strcmp(str\_des,buf->name))

v\_size\_des=buf->size;

if(!strcmp(str\_sou,buf->name))

v\_size\_sou=buf->size;

}

if(!strcmp(m->name,"strcpy") || !strcmp(m->name,"sscanf"))

{

if(v\_size\_des<v\_size\_sou)

flag\_m=1; //明显大于缓冲空间的，一定会溢出

else

flag\_m=3; //没有明显大于，不会溢出

}

else

{

if(!strcmp(m->name,"strncpy") || !strcmp(m->name,"memcpy"))

{

if(v\_size\_des<n)

flag\_m=1;

else

flag\_m=3;

}

else

{

if(!strcmp(m->name,"strcat"))

{

if(v\_size\_des<(v\_size\_des+v\_size\_sou))

flag\_m=1;

else

flag\_m=3;

}

}

}

}

while (bd->b\_next != NULL)

{

bd = bd->b\_next;

flag\_b = 1;

v = f->va\_chain;

while(v->v\_next != NULL)

{

v = v->v\_next;

if(strstr(bd->cont, v->name))

{

flag\_b = 2;

break;

}

}

}

if(flag\_m==1 && flag\_b==0)

over\_flow=1;

if(flag\_m==1 && flag\_b==2)

over\_flow=2;

if((flag\_m==0) || (flag\_m==3 && flag\_b==0))

over\_flow=3;

switch(over\_flow)

{

case 1:

printf("Overflow in line %d:\n",m->row);

Show(pf2,m->row,str\_output);

break;

case 2:

printf("Possibility of overflow in line %d:\n",m->row);

Show(pf2,m->row,str\_output);

break;

case 3:

printf("No overflow!\n");

break;

}

### 整数宽度溢出检测算法

分析整数宽度溢出的原理，对可能造成整数宽度溢出的不同情况进行分析，对每种可以造成整数宽度溢出的原因用一个标志符进行标记，然后最后根据造成整数宽度溢出的情况对这几个标志符进行分类讨论，得到发生溢出的位置；整数宽度溢出检测算法具体实现如下：

while(e->e\_next!=NULL) //赋值语句溢出

{

flag\_e=1;

e=e->e\_next;

strcpy(e\_left,e->left);

strcpy(e\_right,e->right);

while(v->v\_next!=NULL)

{

v=v->v\_next;

if(!strcmp(e\_left,v->name))

type\_flag\_l=v->type\_flag;

if(!strcmp(e\_right,v->name))

type\_flag\_r=v->type\_flag;

}

if(type\_flag\_l<type\_flag\_r)

flag\_e=2;

}

while(bd->b\_next!=NULL) //边界语句溢出

{

bd=bd->b\_next;

flag\_b=1;

v=f->va\_chain;

while(v->v\_next!=NULL)

{

v=v->v\_next;

if(strstr(bd->cont,v->name)!=NULL)

{

flag\_b=2;

break;

}

}

}

while(m->m\_next!=NULL)

{

flag\_m=1;

m=m->m\_next;

strcpy(str\_des,m->des);

while(buf->buf\_next!=NULL)

{

buf=buf->buf\_next;

if(!strcmp(buf->name,str\_des))

flag\_m=2;

}

}

if(flag\_e==2 && flag\_m==2 && flag\_b!=2)

over\_flow=1; //赋值语句左窄右宽，敏感函数和缓冲变量有关，无边界检查（无该语句或检查项与变量无关），一定溢出

if(flag\_e==2 && flag\_m==2 && flag\_b==2)

over\_flow=2; //不一定溢出

if((flag\_e==0 && flag\_b==2)||(flag\_e==2 && flag\_m!=2))

over\_flow=3; //无赋值语句且敏感函数与缓冲变量无关或无敏感函数 或 无赋值语句且有边界检查

switch (over\_flow)

{

case 1:

printf("Overflow in line %d caused by line %d:\n",m->row,e->row);

Show(pf2,e->row,str\_output\_1);

Show(pf2,m->row,str\_output\_2);

break;

case 2:

printf("Possibility of overflow in line %d caused by line %d:\n",m->row,e->row);

Show(pf2,e->row,str\_output\_1);

Show(pf2,bd->row,str\_output\_2);

Show(pf2,m->row,str\_output\_3);

case 3:

printf("No overflow!\n");

}

### 整数计算溢出检测算法

整数计算溢出检测算法原理与整数宽度溢出检测算法大体上是类似的；整数计算溢出检测算法具体实现如下：

while(op->o\_next!=NULL)

{

flag\_op=1;

op=op->o\_next;

strcpy(op\_des,op->des);

while(v->v\_next!=NULL)

{

v=v->v\_next;

if(!strcmp(op\_des,v->name))

des\_type\_flag=v->type\_flag;

}

if(des\_type\_flag<5)

flag\_op=2;

}

while(m->m\_next!=NULL)

{

flag\_m=1;

m=m->m\_next;

strcpy(str\_des,m->des);

while(b->buf\_next!=NULL)

{

b=b->buf\_next;

if(!strcmp(b->name,str\_des))

flag\_m=2;

}

}

if(flag\_op==2 && flag\_m==2)

over\_flow=2;

else

over\_flow=3;

if(over\_flow==2)

{

printf("Possibility of arithmetic overflow in line %d caused by line %d:\n",m->row,op->row);

Show(pf2,op->row,str\_output);

Show(pf2,m->row,str\_output);

printf("\n");

}

else

printf("No arithmetic overflow!\n");

## 2.3 数据结构设计

### 2.3.1 函数体结构体

检测同源性时，构造一个由所有函数体构成的链表，对每个函数体中内容进行比对，从而计算平均相似度。对于每一个函数体定义一个结构体，结构体中包括这个函数体的名字，以及指向该函数体中函数构成的链表的指针以及指向下一个这样结构体的指针。如图

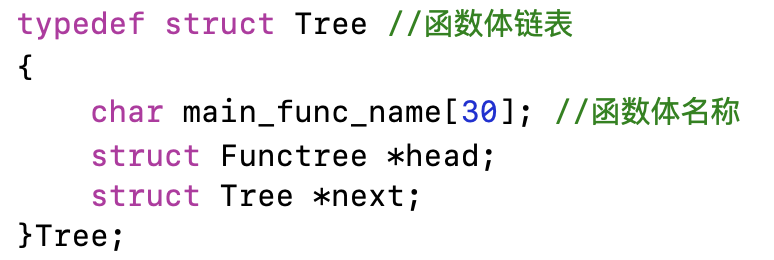


图 1

检测溢出漏洞时，构造一个由所有函数体构成的链表，然后对每个函数体进行分析，从而检测出有溢出可能的代码。对于每一个函数体定义一个结构体，结构体中包括函数体开始的位置，结束的位置以及指向一些与溢出有关的变量或语句构成的链表的指针。栈溢出漏洞检测中，函数体结构体定义如图；堆溢出漏洞检测中，函数体结构体定义如图；整数宽度溢出漏洞检测中，函数体结构体定义如图；整数运算溢出漏洞检测中，函数体结构体定义如图。

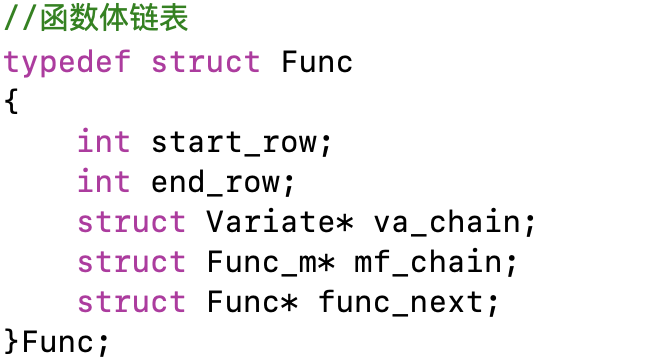


图2

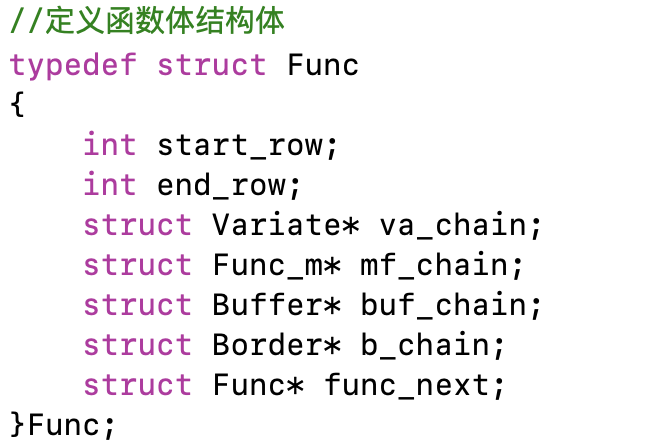


图3

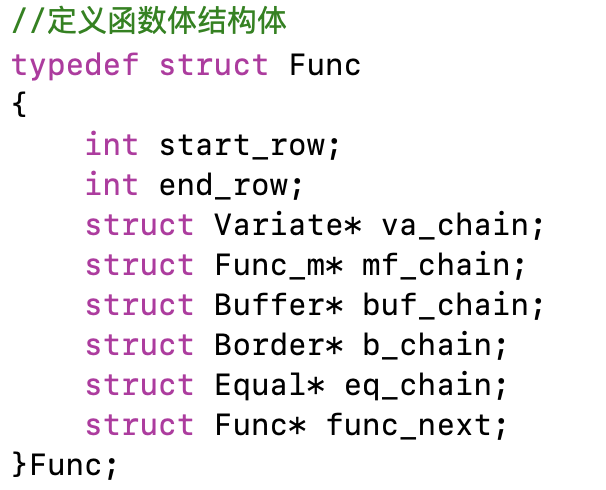


图 4

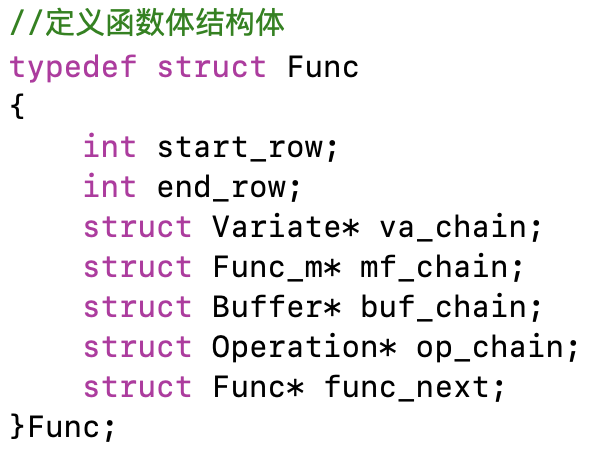


图5

### 2.3.2 函数结构体

检测同源性时，分析每个函数体，将函数体中出现的子函数整理出来，并将他们构造成一条函数链表。对于每个子函数，构造一个结构体存储其参数，结构体中包括函数的返回值、函数名、函数的参数个数以及参数的数据类型，定义如图；

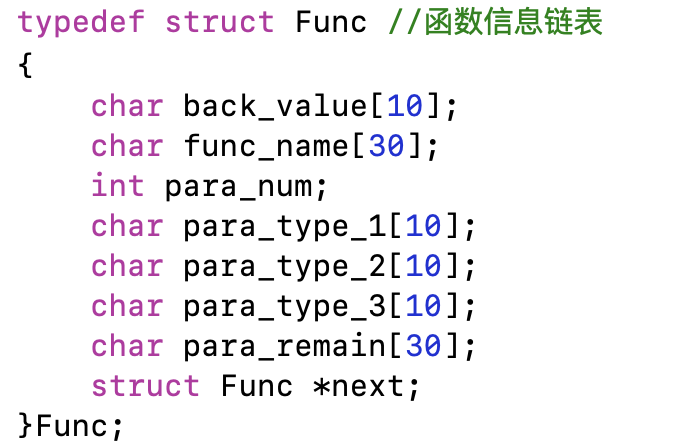


图6

### 2.3.3 敏感函数结构体

检测溢出漏洞时，分析溢出发生原理，可以发现敏感函数是溢出发生的一个原因，对每个函数体，将函数体中出现的所有敏感函数构造一条链表，存储敏感函数的信息，方便查找溢出的位置。对每个敏感函数，定义一个结构体，结构体中包括函数的位置、名字、目的参数以及源参数等信息，定义如下图。



图7

### 2.3.4普通变量结构体

检测溢出漏洞时，需要将每个函数体中出现的变量存储起来，所以对每个普通变量，定义了一个结构体，结构体中包括变量的数据类型、名字以及字节数和位置；方便在溢出检测时通过对变量链表的遍历获取敏感函数中出现的目的参数和源参数的信息，以及在赋值语句中出现的左值与右值的信息，从而判断是否有溢出的可能。栈溢出检测和堆溢出检测中普通变量结构体如图，整数宽度溢出检测和整数运算溢出检测中普通变量结构体如图。

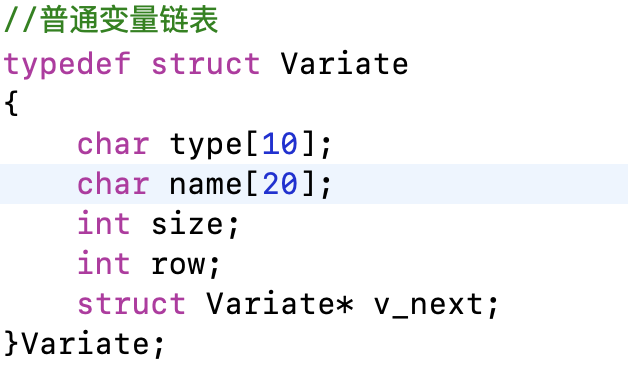


图8

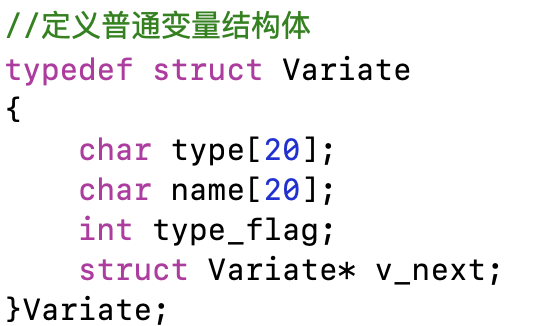


图9

### 2.3.5 缓冲区变量结构体

检测溢出漏洞时，分析溢出发生的原理，可以发现缓冲区变量的使用也是造成溢出发生的一个方面；对缓冲区变量，定义一个结构体，结构体中包括缓冲区变量的数据类型、名字以及空间大小等信息，从而在判断溢出的时候可以遍历整个链表，分析使用过程中是否发生了溢出。缓冲区变量结构体定义如图

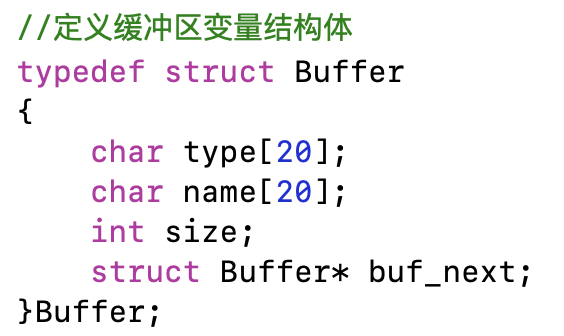


图10

### 2.3.6 边界检查语句结构体

对于边界检查函数，若语句中包含了缓冲区变量，就有可能发现溢出。所以对于边界检查语句定义一个结构体，存储该边界检查语句的信息，然后在分析溢出情况的时候可以遍历缓冲区变量链表判断该语句是否含有缓冲区变量，从而判断是否有溢出的可能性。结构体中包括边界检查语句的位置、语句类型以及内容等，具体定义如下图。

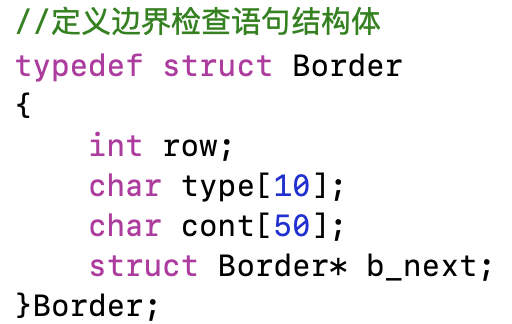


图 11

### 2.3.7 赋值语句结构体

对于整数宽度溢出，分析其原理可以发现在赋值的过程中容易发生溢出，尤其是长整数赋值给短整数最容易发生溢出。对于赋值语句同样可以定义一个结构体存储其信息，并构造链表，从而可以在遍历函数体分析溢出的时候判断该赋值语句是否发生溢出；该结构体中包括该语句的位置、左值和右值，定义的赋值语句结构体如下图。

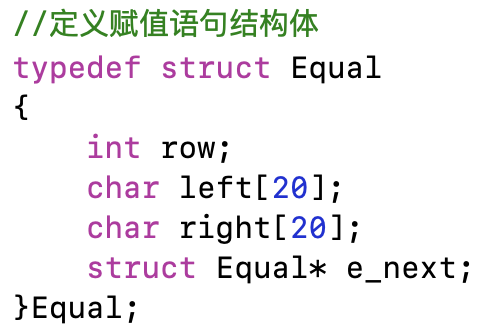


图12

### 2.3.8 运算语句结构体

在整数运算溢出检测中，需要对每个运算语句进行分析，一个运算语句包括目的参数和两个进行运算的源参数。所以对于运算语句可以定义这样一个结构体进行描述，结构体中包括语句的位置，方便确定溢出位置，进行运算的两个源参数和运算结果目的参数；运算语句结构体定义如下图。

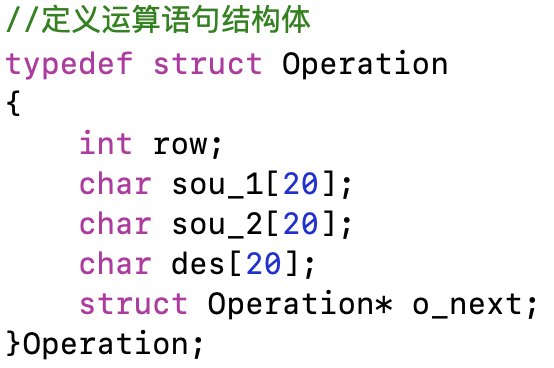


图13

## 2.4 样本库设计

### 2.4.1 同源性检测样本库

同源性检测样本库中样本数大于等于 50 个，每个代码行数不少于100行,包含1-100行相同代码；如图所示。

### 2.4.2 漏洞检测样本库

漏洞检测样本库中样本数大于等于 50 个，每个代码行数不少于100行, 每种漏洞至少一个；如图所示。

## 2.5 关键问题及其解决方法

### 2.5.1 漏洞检测原理

溢出漏洞的检测可以分为三部分：与缓冲区有关的敏感函数的检测，引起溢出的操作语句的检测，防止溢出操作的检测。

对于敏感函数，根据函数结构分情况讨论，分析敏感函数的源参数、目的参数和其它参数，若源参数的缓冲区大小与n运算后大于目的参数的缓冲区的大小，则会发生溢出，反之不会。

对于运算语句，若存放运算结果的变量宽度小于操作数的宽度，就有可能发生溢出。

对于赋值语句，若等号左边的变量的宽度小于右边，就有可能发生溢出。

对于边界检查函数，若语句中包含了缓冲区变量，就有可能发现溢出，制止进一步的调用敏感函数操作。

### 2.5.2 CFG的存储问题

为便于输出CFG和进行后续的比对，采用链表这种结构，而一条链表无法直观得表示代码中所有的函数调用关系，所以采用十字链表这种结构。

每个函数体建立一条链表，存放函数体中调用的函数的名称。每份代码建立一条链表，存放各个函数体链表的头指针。

# 程序设计

## 3.1 系统各功能程序设计

### 3.1.1利用字符串匹配进行同源性检测

1. 程序设计思路

（1）假设有两份代码a和b.

（2）对a和b分别进行预处理，便于后期比对。去除预编译语句、C语言标准注释、C++标准注释。

（3）逐行读取代码a，用从a中读取的一行与b中每一行进行比对，如果在b中找到相同的字符串则计数器加一。

（4）将a中与b中相同的字符串的数量与a中字符串的数量求比例即为所求匹配度。

### 3.1.2利用CFG图进行同源性检测

1．程序设计思路

(1) 假设有两份代码a和b.

(2) 对a和b分别进行预处理，便于后期比对。去除预编译语句、C语言标准注释、C++标准注释、空行和行前空格。

(3) 分别建立a和b和函数调用树A和B。

(4) 遍历树A和B，找到函数名相同的函数体，在该函数体内统计函数名相同的子函数的个数n和函数体内全部子函数的个数N。

(5) 计算n/N,计算结果为这两个函数体的相似度rate\_line，并将结果存入数组rate,其中数组大小和代码中函数体的个数相同。

(6) 重复（4）和（5）的操作，直至树A或B有一个为空。

(7) 计算数组rate中所有元素的平均值，得到树A和B的相似度，即代码a和b的函数调用关系相似度。

(8) 在建立函数调用树时，先扫描全文，找到函数定义或声明，分析函数，提取出函数名、函数返回值的类型、参数的个数及各参数的类型，并将这些信息存入函数链表F，方便后序查找。

(9) 再扫描全文，找到函数体，并为每个函数体建立一条链表f，用来存放函数体内子函数的名称。在函数体内，扫描每条语句，遍历函数链表F查看函数体内有无调用子函数，并将调用的子函数的名称存入链表f.

(10) 重复上述（8）、（9）操作，直至文件结尾。

2.流程图

基于CFG图进行同源性检测程序的流程图如下图所示，函数结构体构成的链表的建立的流程图如下图所示。

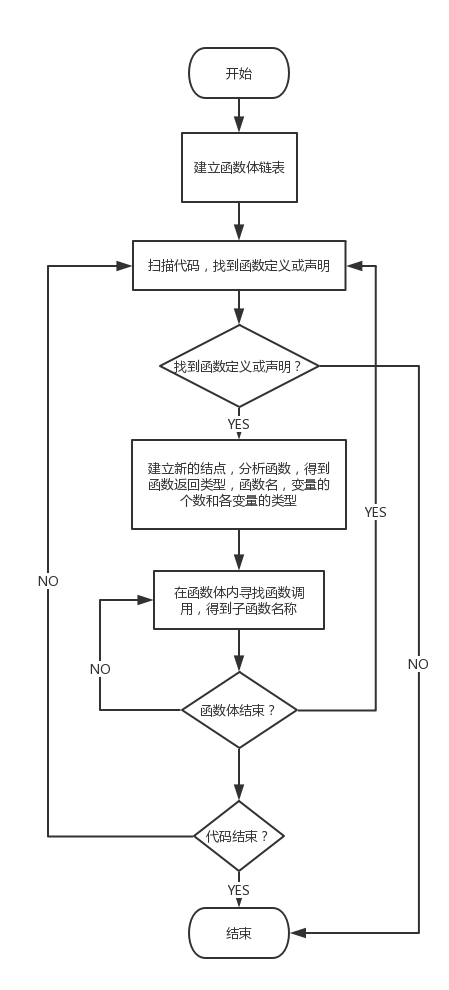
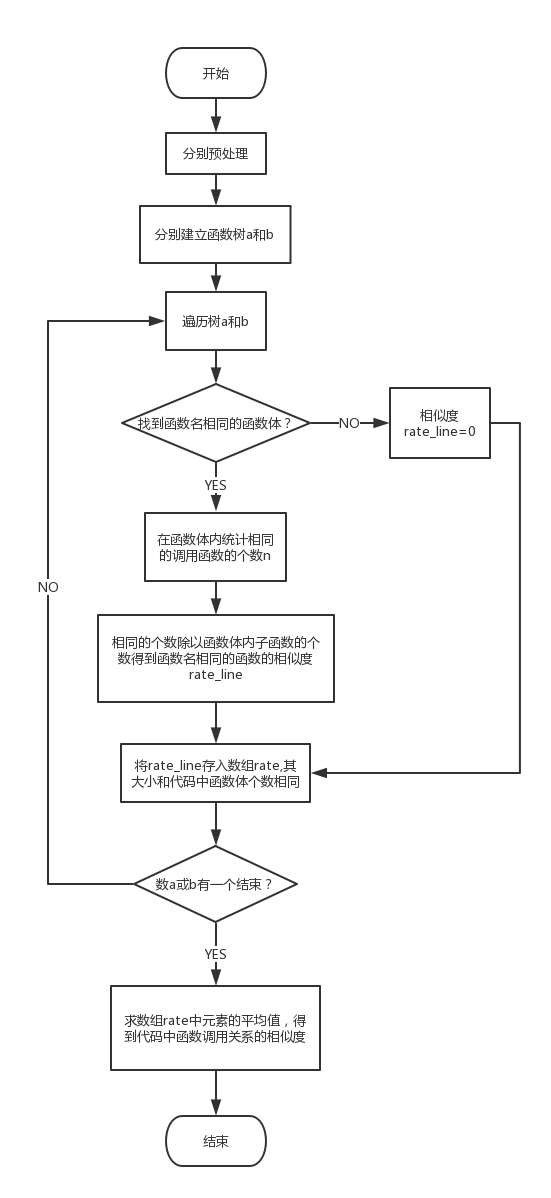
****

图 14 图 15

### 3.1.3 栈溢出漏洞检测

1. 程序设计思路

(1) 假设需要判断溢出的代码为a.

(2) 对a进行预处理，便于后序操作。去除预编译语句、空行、C语言标注释、C++标准注释和行前空格。

(3) 扫描a，建立函数体链表F，用于存放函数链表。

(4) 找到函数体后，为该函数建立存放敏感函数的链表和存放变量的链表f，并记录函数体开始的位置。

(5) 扫描函数体，分析变量的名称、类型和所占缓冲区的大小，并将相关信息存放入变量链表。

(6) 扫描函数体，找到敏感函数，分析得到敏感函数的名称、源参数、目的参数和其它参数n,并将相关信息存放入敏感函数链表。

(7) 重复（5）、（6）操作，直至函数体结束，并记录函数体结束的位置。

(8) 函数分析结束，根据分析所得数据进行溢出判断。

(9) 遍历函数体链表F，在每个函数体内，遍历敏感函数链表，若函数体内无敏感函数，则标志位置为3，退出该函数体；否则，得到敏感函数的名称、源参数 、目的参数和其它参数的名称。遍历变量链表，得到源参数、目的参数所占缓冲区的大小des\_size和sou\_size，根据敏感函数的名称，分情况对sou\_size和n进行运算，若运算结果res>des\_size,则一定溢出，标志位置为1，否则置为3，不会发生溢出。

(10) 给出判定结果，并输出溢出发生的位置和导致溢出的代码。

(11) 重复进行（9）、（10）操作，直至F为空。

2. 流程图

栈溢出漏洞检测程序的总流程图如下图所示，对函数体内溢出可能性的检测的程序流程图如下图所示。

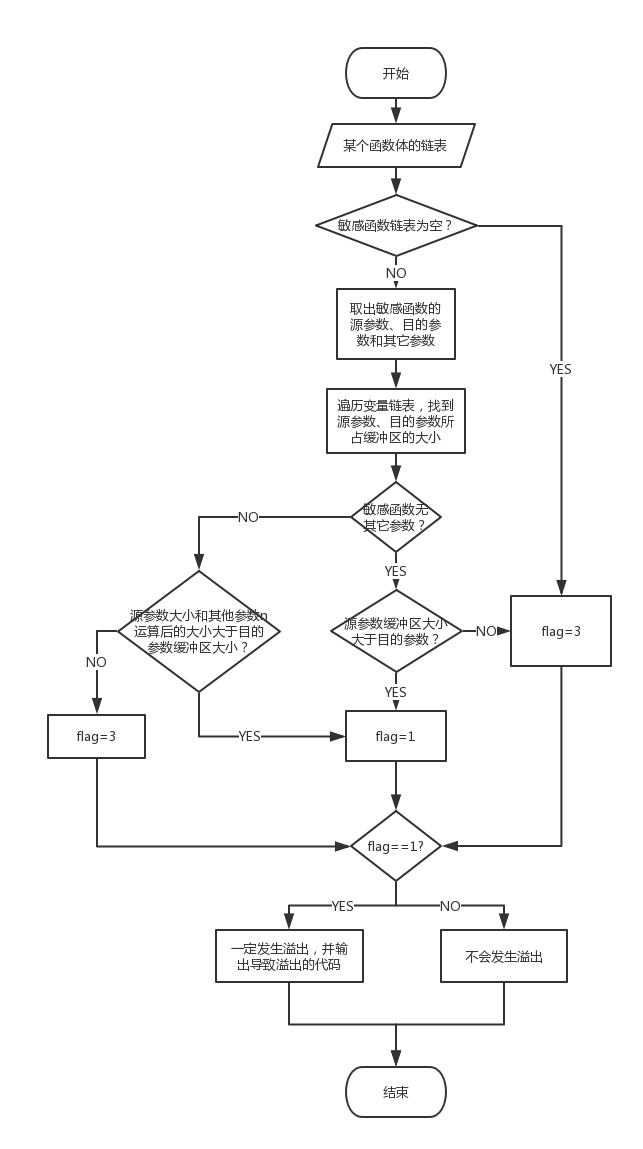
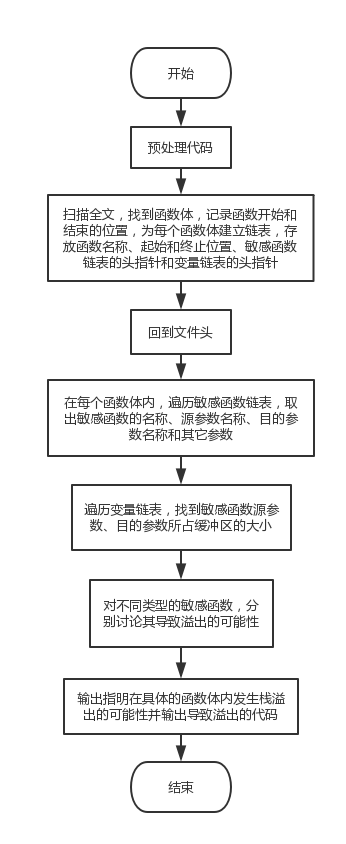


图16 图17

### 3.1.4 堆溢出漏洞检测

1. 程序设计思路

(1) 假设要判断的代码为a.

(2) 对a进行预处理，便于后序操作。去除预编译语句、C语言标准注释、C++标准注释、空行和行前空格。

(3) 扫描a,建立函数体链表F，用于存放函数链表。

(4) 找到函数体，为该函数体建立函数链表f，用于存放函数名称，起始、终止位置、普通变量链表的头指针、缓冲区变量链表头指针、敏感函数链表头指针和边界检查语句头指针。

(5) 扫描函数体，在函数体内寻找普通变量和缓冲区变量，分析得到变量名称、类型和所占缓冲区大小，并将信息存入对应链表。

(6) 扫描函数体，在函数体内寻找敏感函数，分析得到敏感函数的名称、源参数sou、目的参数des和其它变量n,并将以上信息存入链表。

(7) 扫描函数体，在函数体内寻找 边界检查语句（if语句），取出语句的内容并存入链表。

(8) 重复（4）、（5）、（6）、（7）操作，直至文件结束。

(9) 函数分析结束，根据分析所得数据进行堆溢出检测。

(10) 遍历函数体链表F，对于一条函数链表f，遍历敏感函数链表，若无敏感函数，至标志位flag\_m=0并退出该函数体；否则，取出敏感函数的名称、源参数sou、目的参数des和其它参数n.遍历缓冲区变量链表，取出sou、des的大小sou\_size和des\_size.根据敏感函数的类型，分情况讨论sou\_size,des\_sze和n的大小关系。若sou\_size与n的运算结果明显大于des\_size,则flag\_m=1,一定溢出，否则flag\_m=3,不会溢出。

(11) 遍历f中的边界检查语句链表，若无边界检查语句，则flag\_t=0;否则，取出语句内容，若其与普通变量有关，则至flag\_t=2,否则flag\_t=1.

(12) 对flag\_m和flag\_t整合分析：若flag\_m!=1,则不会发生溢出；若flag\_m==1且flag\_t!=2,则发生溢出的可能性较大；若flag\_m==1且flag\_t==2,则发生溢出的可能性较小；若flag\_m==1且flag\_t==0,则一定会发生溢出。

(13) 给出溢出可能性大小的判断并输出溢出位置和导致溢出的代码。

2. 流程图

堆溢出漏洞检测程序的总流程图如下图所示，对函数体内溢出可能性的检测的程序流程图如下图所示。

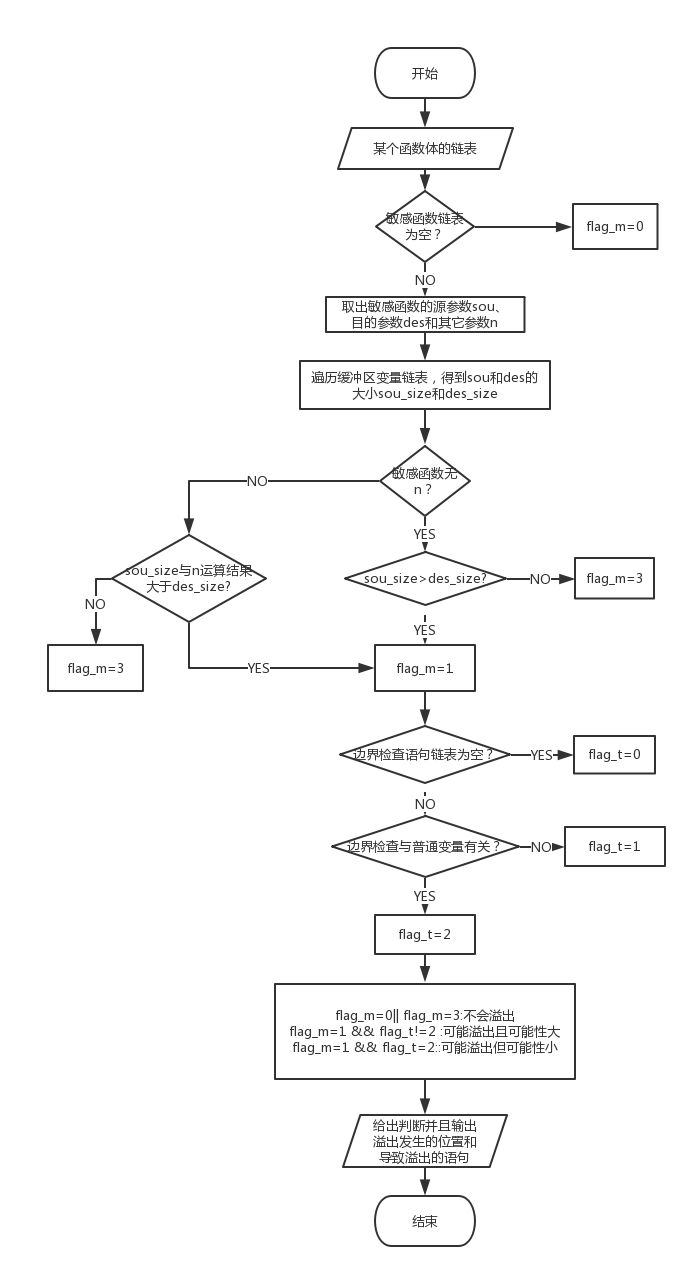
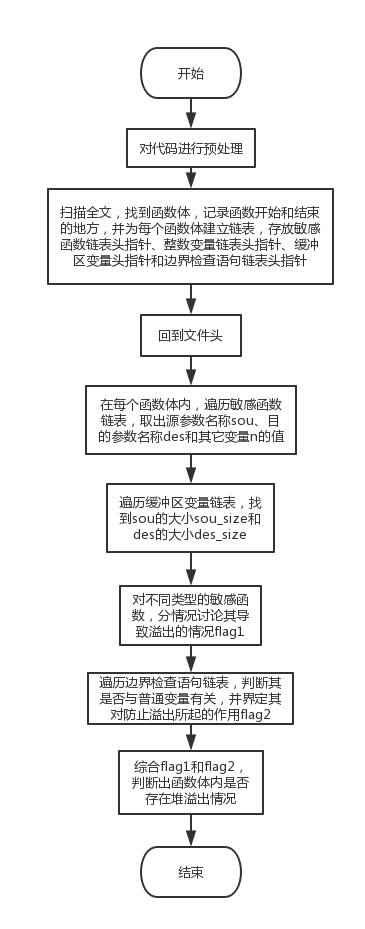
****

图18 图19

### 3.1.5 整数宽度溢出漏洞检测

1. 程序设计思路

(1) 假设需要检测的代码为a。

(2) 对a进行预处理，便于后序操作。去除预编译语句、C语言标准注释、C++标准注释、空行和行前空格。

(3) 建立函数体链表F，用来存放函数链表的头指针。

(4) 扫描文本，找到函数体，为该函数体建立函数链表f，用来存放函数名称、函数体起始和终止位置、赋值语句链表头指针、敏感函数链表头指针、普通变量链表头指针、缓冲区变量链表头指针和边界检查语句链表头指针。

(5) 扫描函数体，分析变量，获得整数变量的名称、类型，并根据其宽度大小设定标志位1~6,1为最窄宽度，6为最大宽度。

(6) 扫描函数体，获得缓冲区变量的名称，并将其存入链表。

(7) 扫描函数体，分析赋值语句，得到等号左边变量的名称vl和右边的名称vr.

(8) 扫描函数体，分析边界检查语句（if语句）。

(9) 重复以上（4）~（8），直至文件结束。

(10) 遍历函数体链表F，对于一条函数链表f，扫描赋值语句链表，取出左值vl和右值vr。遍历普通变量链表，得到vl和vr的宽度标志位flag\_l和flag\_r，若flag\_l<flag\_r,则flag\_e=1,否则为3.

(11) 遍历敏感函数链表，取出源参数sou和目的参数des，遍历缓冲区变量链表，判断sou和des与缓冲区变量是否有关，有关，flag\_m=1,否则为3.

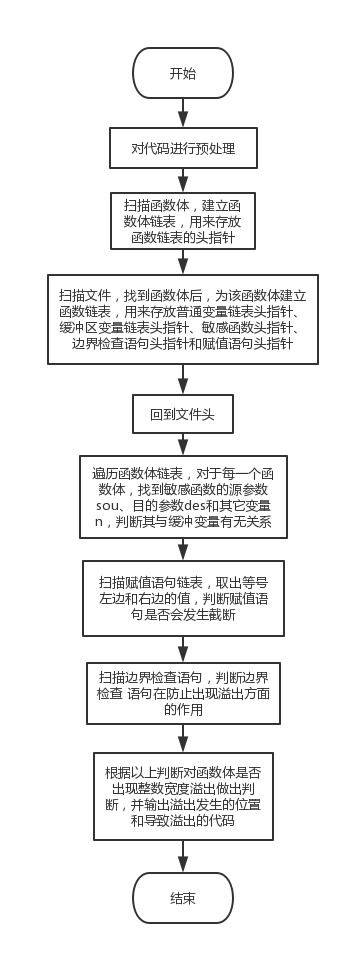
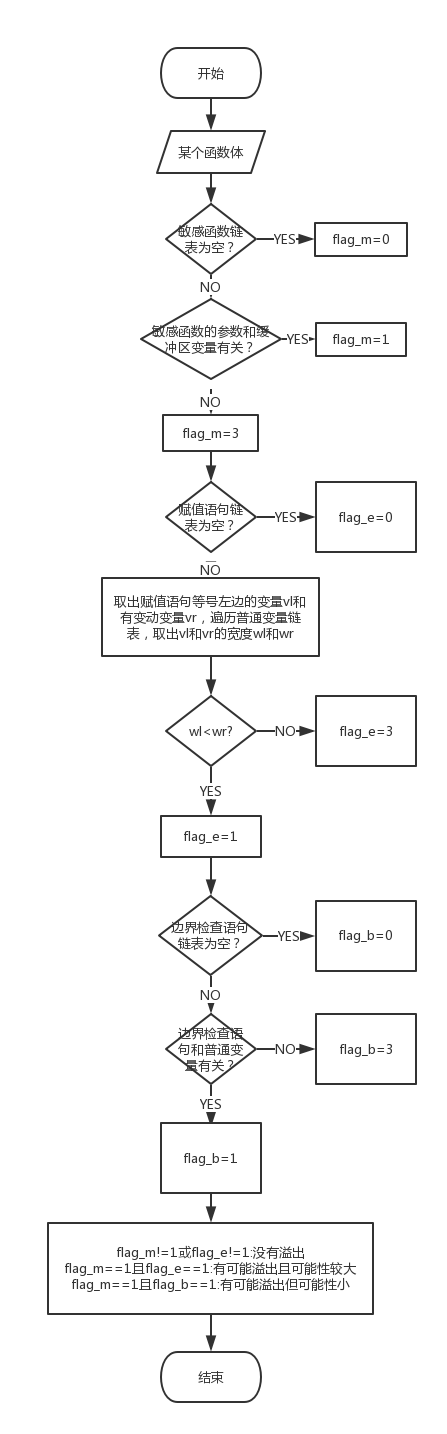
(12) 遍历边界检查语句链表，取出语句内容，判断语句与普通变量是否有关，有关，flag\_b=1,否则为3.

(13) 根据以上判断给出结论：若flag\_m!=1或flag\_e!=1，不可能溢出；若flag\_m==1且flag\_e==1且flag\_b!=1,则有很大可能溢出；若flag\_m==1且flag\_e==1且flag\_b==1，溢出可能性减小。

(14) 输出判断结果，并输出溢出发生位置和导致溢出的代码。

2. 流程图

整数宽度溢出漏洞检测程序的总流程图如下图所示，判断某个函数体内有无整数宽度溢出的程序流程图如下图所示。

** **

**图 20 图21**

### 3.1.6 整数运算溢出漏洞检测

1. 程序设计思路

(1) 假设需要检测的代码为a。

(2) 对a进行预处理，便于后序操作。去除预编译语句、C语言标准注释、C++标准注释、空行和行前空格。

(3) 建立函数体链表F，用来存放函数链表的头指针。

(4) 扫描文本，找到函数体，为该函数体建立函数链表f，用来存放函数名称、函数体起始和终止位置、运算语句链表头指针、敏感函数链表头指针、普通变量链表头指针和缓冲区变量链表头指针。

(5) 扫描函数体，分析变量，获得整数变量的名称、类型，并根据其宽度大小设定标志位1~6,1为最窄宽度，6为最大宽度。

(6) 扫描函数体，获得缓冲区变量的名称，并将其存入链表。

(7) 重复以上（4）~（6）直至文件结束。

(8) 遍历函数体链表F，对于一条函数链表f，若其运算语句链表为空，则置flag\_a=0且退出该函数，否则，取出存放运算结果的变量名res，遍历普通变量链表，取出res的宽度标志位，若其>=4，则flag\_a=3,否则为1。

(9) 遍历敏感函数链表，判断敏感函数的参数是否与缓冲区变量相关，若相关，则flag\_m=1,否则为3.

(10) 根据以上判断给出结论：若flag\_a!=1或flag\_m!=1，不可能溢出；若flag\_m==1且flag\_a==1,则可能发生溢出。

(11) 输出溢出发生的位置和导致溢出的代码。

2. 流程图

整数运算溢出漏洞检测程序的总流程图如下图所示，判断某个函数体内有无整数运算溢出的程序流程图如下图所示。

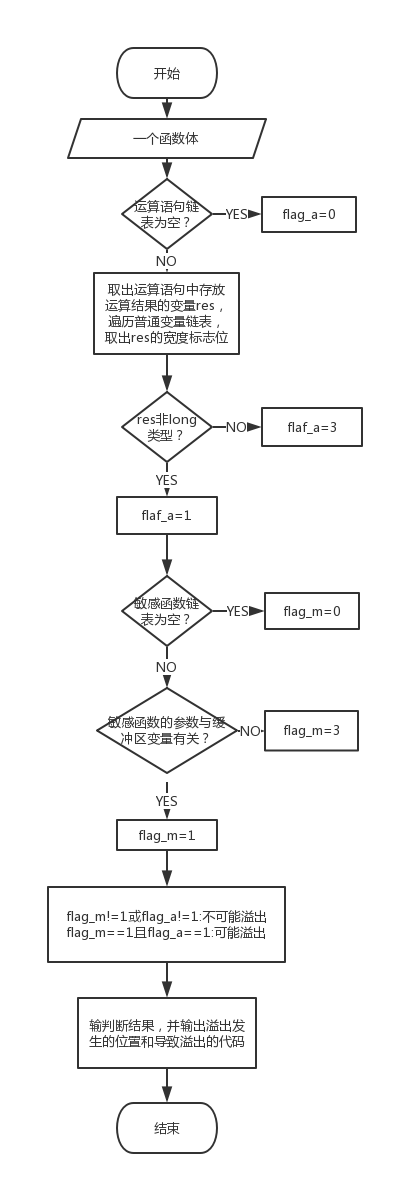
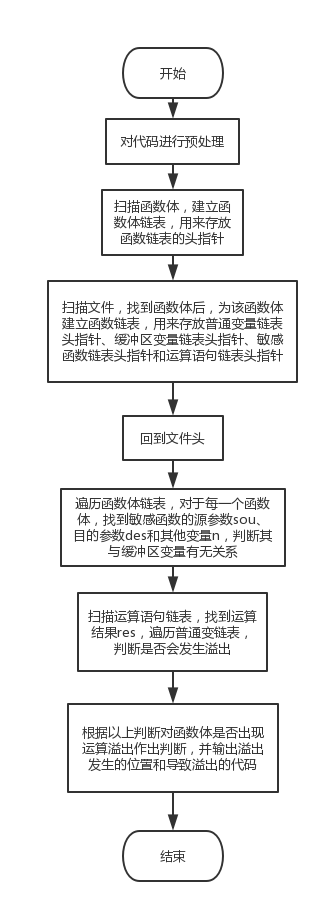
****

图22 图23

# 4 系统测试

## 4.1利用字符串匹配进行同源性检测

利用字符串匹配进行同源性检测，测试文件为1.cpp和2.cpp, 测试结果如下：

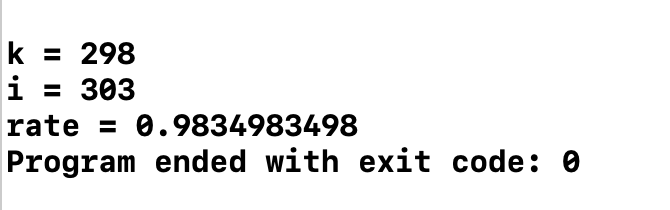


图24

## 4.2利用CFG图进行同源性检测

对基于CFG图进行同源性检测程序，输入需要比对的文件名字以及预处理后存储文件的文件名，程序会将生成的CFG图输出，并输出相似度。测试结果如下图：



图 25

## 4.3栈溢出漏洞检测

对于栈溢出漏洞检测程序，需要输入需要检测的文件名字以及预处理后存储文件的文件名，程序会将生成的CFG图，并输出溢出可能发生的地方的为位置和代码。测试结果如下图

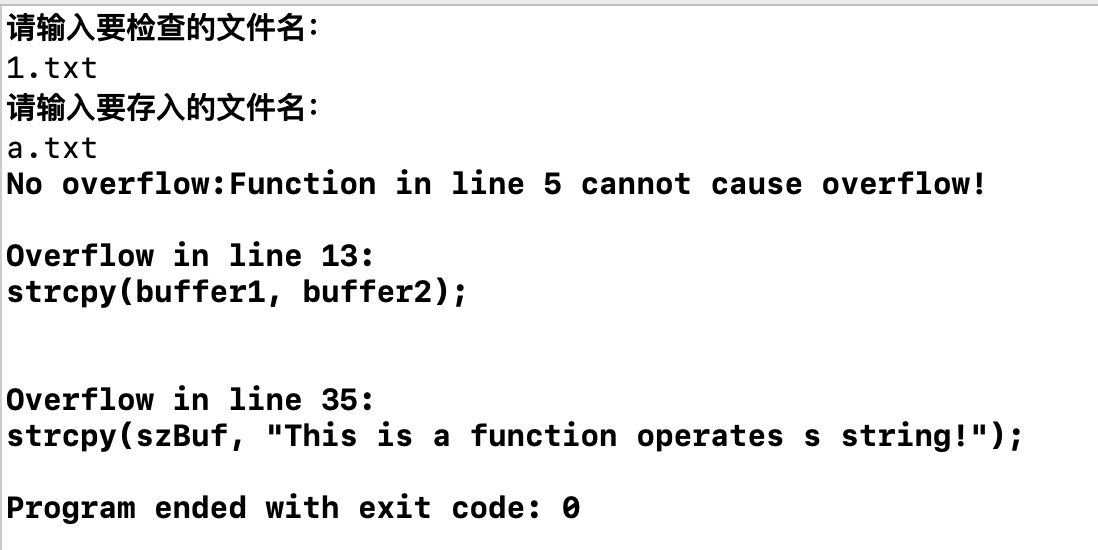


图 26

## 4.4堆溢出漏洞检测

对于堆溢出漏洞检测程序，需要输入需要检测的文件名字以及预处理后存储文件的文件名，程序会将生成的CFG图，并输出溢出可能发生的地方的为位置和代码。测试结果如下图

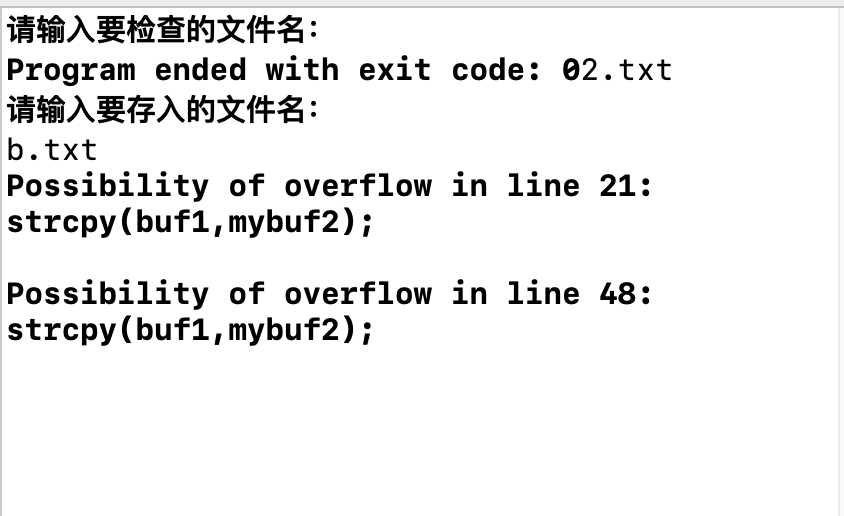


图 27

## 4.5整数宽度溢出漏洞检测

对于整数宽度溢出漏洞检测程序，需要输入需要检测的文件名字以及预处理后存储文件的文件名，程序会将生成的CFG图，并输出溢出可能发生的地方的为位置和代码。测试结果如下图

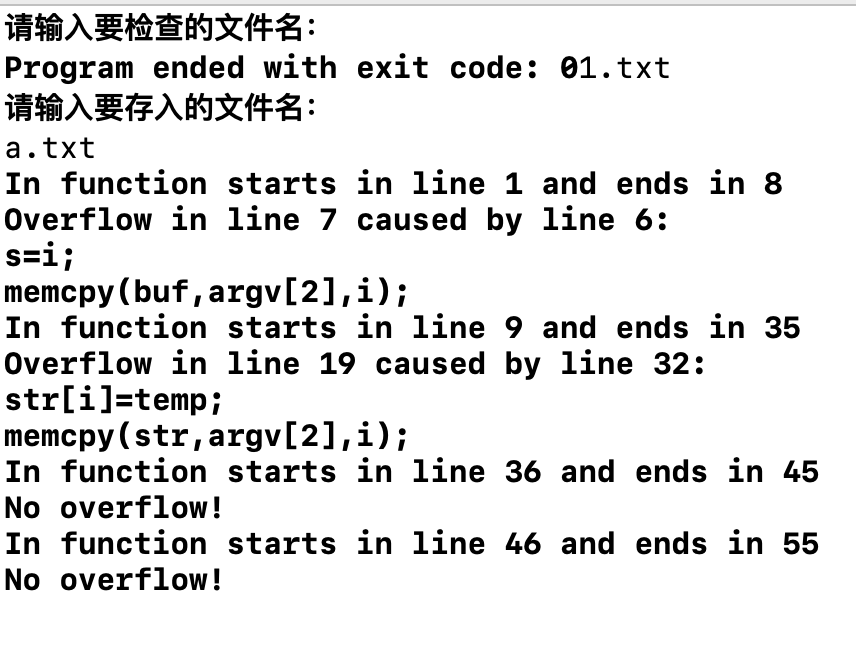


图 28

## 4.6整数运算溢出漏洞检测

对于整数运算溢出漏洞检测程序，需要输入需要检测的文件名字以及预处理后存储文件的文件名，程序会将生成的CFG图，并输出溢出可能发生的地方的为位置和代码。测试结果如下图

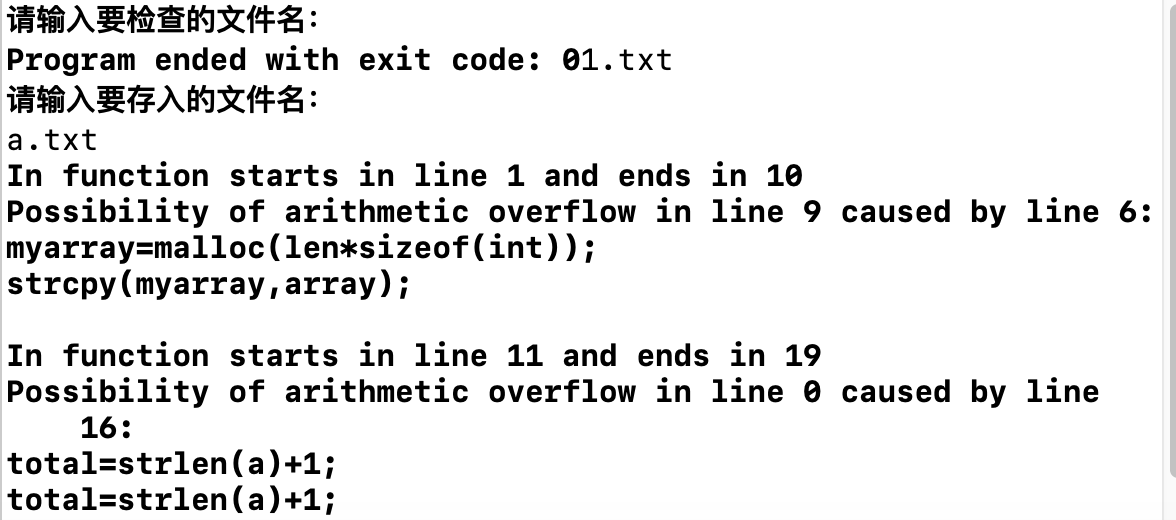


图29

## 4.7同源性样本库检测

使用同源性样本库实现基于CFG图检测代码同源性的测试程序测试结果如图。



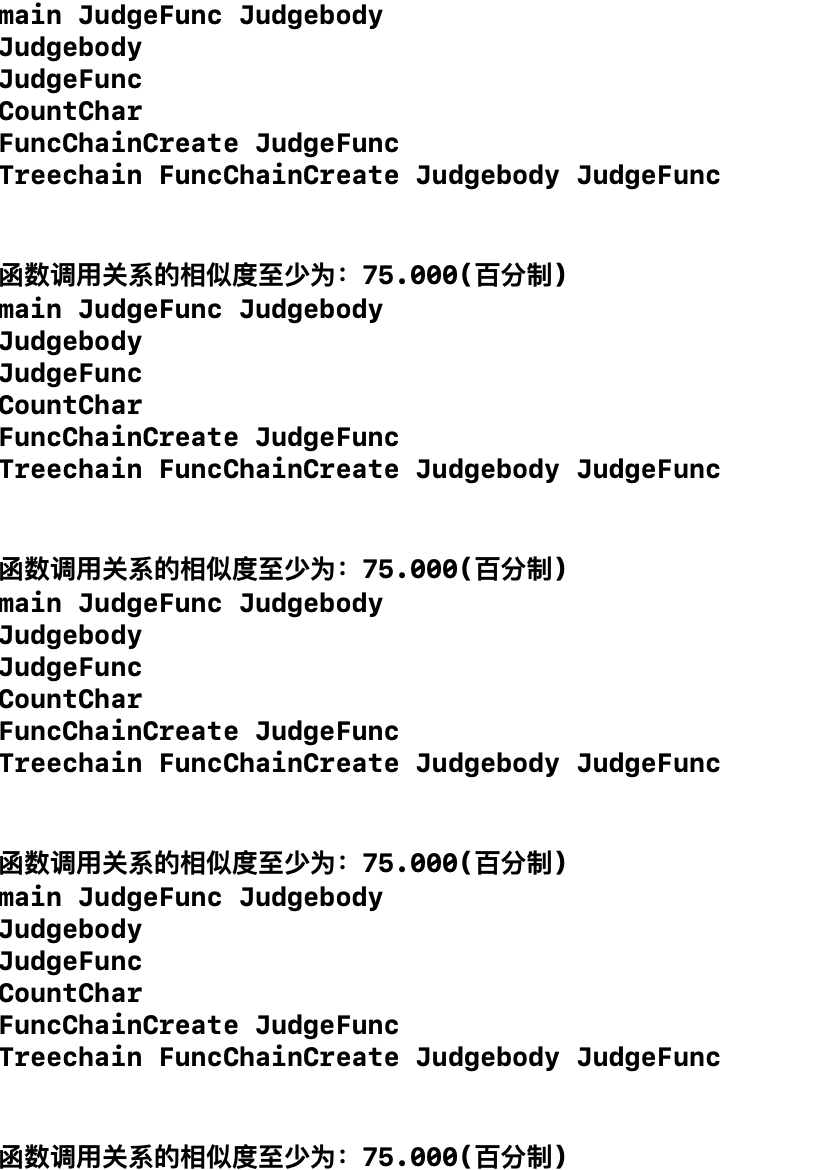
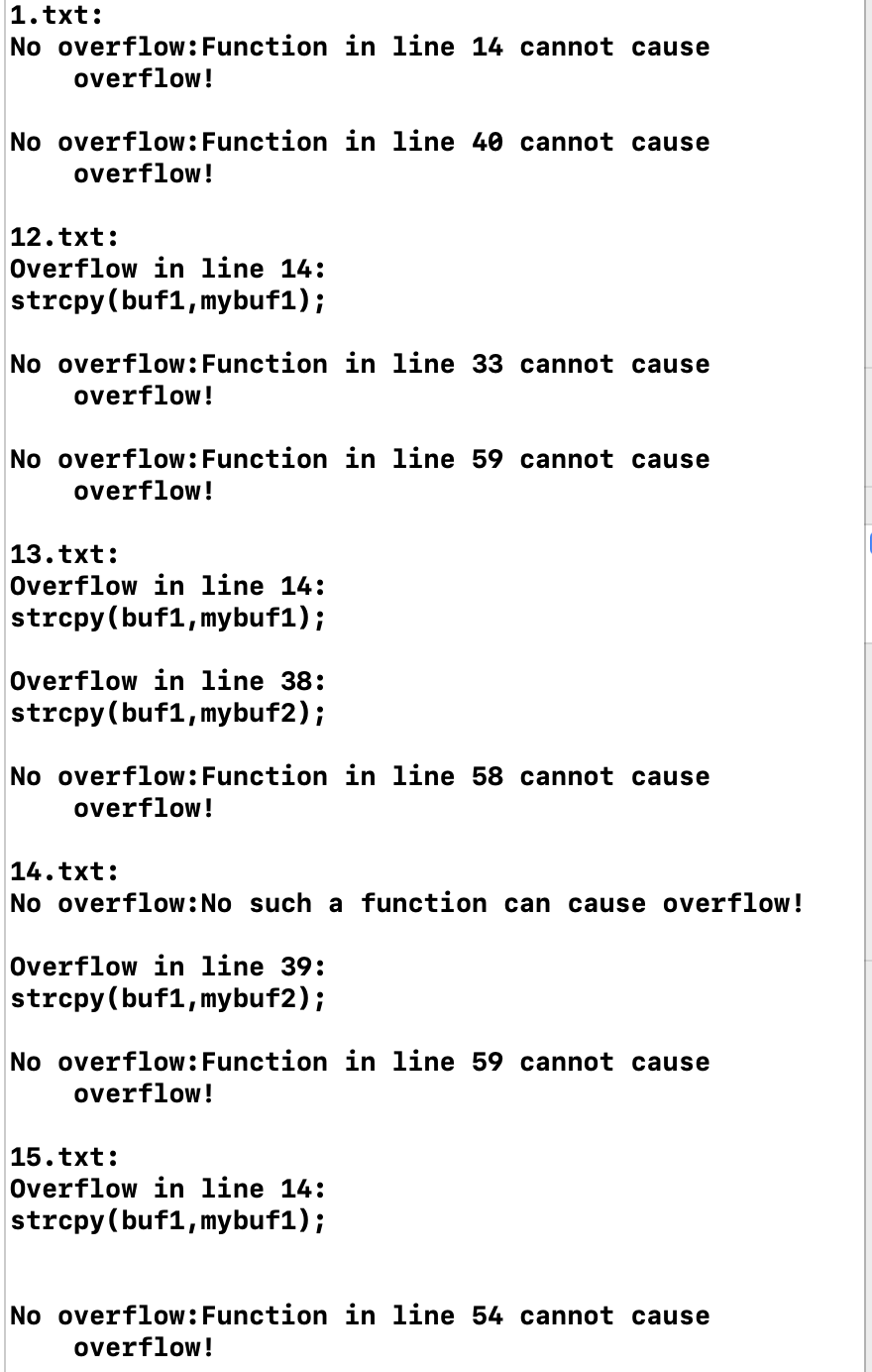


图 30

## 4.8漏洞样本库检测

使用漏洞样本库检测代码栈溢出漏洞的测试程序测试结果如图。



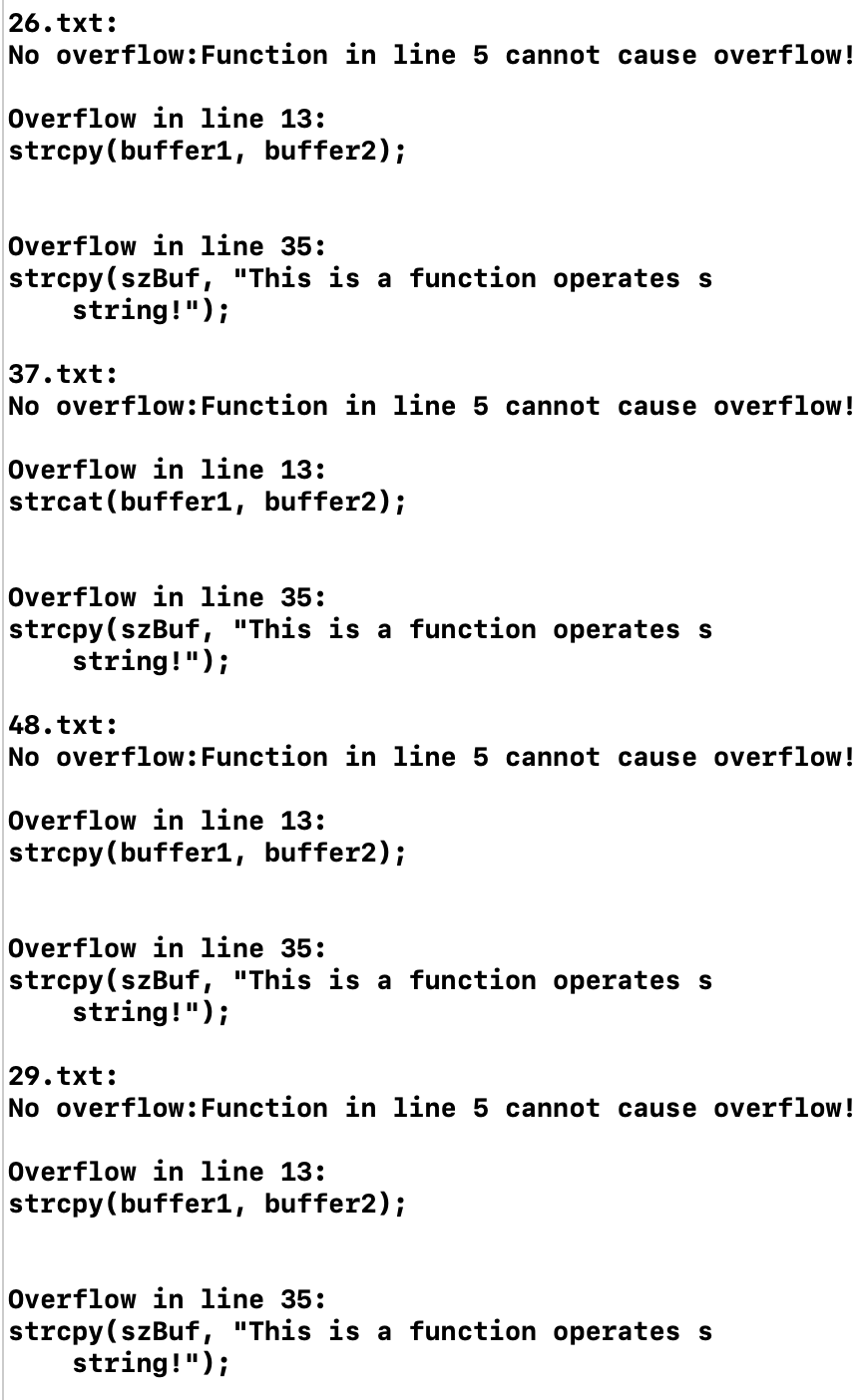


图31

# 5 总结与体会

在课设开始之前看到这个课题还挺有信心的。上学期通过《软件安全》课程了解了漏洞的原理，自己也做了很多功课，所以我认为自己完成课设是没有问题。但是真正动手写代码时会出现各种各样的问题：

1. 利用CFG比较时，刚开始我不知道如何存储cfg，一层链表无法表示多个函数体内的函数调用情况，最后我采用了十字链表这种结构。

2. CFG创建完成后，如何比较又是个问题。在调试过程中我发现了进行第一次遍历之后，后续的遍历不能从头开始，于是我在遍历之前设置变量保留了链表头，解决了这个问题。

3. 完成第一个漏洞检测花费的时间和精力是最多的，如何分析变量，如何分析敏感函数，如何判断敏感函数会不会引起溢出等等都需要考虑。完成了第一个漏洞检测时候，剩下的思路是一脉相承的，只需做少许改动即可。

总而言之，完成课设的过程是痛苦又甘甜的，因为这个过程我收获了很多经验：

1. 在动手写代码之前，尽可能全面地考虑好框架，在写的过程中不断地修改小细节。

2. 不能只求速度。在完成一个大工程的过程中不能依赖完成之后的整体调试，每完成一个部分就编写测试程序进行调试，确认无误后再进行下面的工作，这样到最后出现的问题就会少很多，可以减少心理负担，因为有时候改bug的时间比写程序的时间还多。

3. 遇到问题不能气馁。在完成课设时无疑会遇到各种各样的问题，不能气馁，要从多角度找寻解决问题的方法。

4. 要多和周围同学交流。