

基于模型检测的内存泄露静态测试方法

付晓毓, 朱 利, 顾 伟

(西安交通大学 软件学院, 陕西 西安 710049)

摘 要: 在 C 等支持动态内存分配的语言中, 指针的使用带来的内存泄露是导致系统性能降低的重要问题. 针对现有处理方法的不足, 文中设计了一种基于模型检测技术的内存泄露静态检测方法. 该方法通过建立基于指针属性的内存泄露漏洞模型, 将相应约束断言插桩进源代码, 然后利用模型检测工具验证断言的可达性来判断内存泄露. 实验结果表明, 该方法是有效的和精确的.

关键词: 模型检测; 内存泄露; 代码插桩; 静态测试

中图分类号: TP311

文献标识码: A

文章编号: 1000-7180(2010)10-0170-04

Static Detection to Memory Leak Based on Model Checking

FU Xiao-yu, ZHU Li, GU Wei

(School of Software, Xi'an Jiaotong University, Xi'an 710049, China)

Abstract: In C language that support dynamic memory operations, the usage of pointer brings about memory leak which is the key factor that causes system performance degradation. For the inadequacy of existing approaches, a method of static detection to memory leak based on model checking was proposed in this paper. By modeling memory leak vulnerabilities based on the attributes of a pointer, the method instrument the source code with corresponding constraint assertions, and then verify the reachability of these assertions by model checking tools. Experimental results show that the method is effective and accurate.

Key words: model checking; memory leak; code instrumentation; static detection

1 引言

C 程序设计语言中提供的动态内存机制^[1]为灵活地进行程序设计提供了方便, 但也会引起大量的内存故障. 内存泄漏故障通常会引起严重问题, 它们会逐渐消耗内存资源, 降低系统的性能, 在程序长时间运行时更为严重^[2-3].

基于抽象-验证-细化范例的模型检测技术^[4]是一种有限状态系统的形式化验证方法, 它通过算法穷尽地搜索系统的有限状态空间, 检查系统(模型)的每个状态是否满足预期性质. 模型检测技术能够很好地利用程序中的控制流和信息流进行更

为精确的程序属性分析; 并且其自动化程度高, 在硬件系统和通信协议验证等方面的应用中取得了巨大成功. 通过对 Blast^[5]、Magic^[6]等模型检测工具的研究与使用, 发现单纯使用这类工具还不能对内存泄露漏洞进行检测.

针对上述问题, 文中首先对内存泄露漏洞类型进行分析, 接着对指针及其更新操作进行抽象, 建立基于指针属性的内存泄露漏洞模型——指针属性的约束条件. 然后, 将指针属性的更新操作和约束条件的断言插入到相应的程序代码中, 这样便将内存泄露漏洞的查找问题转化为代码中特定位置断言的判定问题. 最后, 利用模型检测工具对特定位置的断言

收稿日期: 2009-08-12; 修回日期: 2009-10-18

基金项目: 国家自然科学基金项目(60673087); 国家“八六三”计划项目(2003AA1Z2610)

进行判定,如果判定结果为不正确,则表示该位置存在内存泄露漏洞。

2 内存泄露漏洞类型分析

常说的内存泄露是指堆内存的泄露。堆内存是指程序从堆中分配、大小任意、使用完后必须显式释放的内存。应用程序一般使用 `strdup`、`malloc`、`calloc`、`realloc` 等操作从堆中分配一块内存,使用完后程序必须负责调用 `free` 释放该内存块,否则,这块内存就不能被再次使用,即出现内存泄露。

根据堆内存释放产生的不同结果,可将内存泄露分为如下三类:

(1) 未释放动态申请的内存空间

使用 `strdup`、`malloc`、`calloc`、`realloc` 等操作动态申请了堆内存空间,但最后没有对其进行相应的空间释放,例如:

```
int main ( )
{
    int a = 0 ;
    int * b = (int *) malloc (sizeof (int) * 10);
    if (a != 0) free (b);
    return 0;
}
```

(2) 重复释放同一内存空间

这类漏洞多数是由于编译器没有对指针进行别名分析引起的,例如:

```
int main ( ) {
    char * p, * q;
    p = (char *) malloc (sizeof (char) * 10);
    strcpy (p, "hello");
    q = p;
    strcpy (p, "world");
    free (p);
    free (q);
    return 0;
}
```

(3) 部分释放动态申请的内存空间

释放的内存空间和申请的空间大小不一样,例如:

```
int main ( )
{
    int i, p;
    p = (int *) malloc (sizeof (int) * 10);
    for (i=0; i<6; i++) p++;
    free (p);
    return 0;
}
```

通过以上分析不难发现堆内存泄露的发生与指向堆内存的指针的变化密切相关。所以,要建立内存泄露漏洞模型,必须首先对内存及其上的更新操作进行抽象。Pande 等人指出,当只允许使用一级指针时,指针指向问题具有多项式复杂度的解法^[7],所以,文中先讨论存在一级指针的情况。

3 建立内存泄露漏洞模型

3.1 对指针属性及其更新操作的抽象

定义 1 指向动态分配的堆内存的指针集合 P 。令 p, q 表示指向任一堆内存的指针,则 $p, q \in P$ 。

定义 2 表示指针 p 的变化的变量 $p_loc, p_loc \in Z^+$ 。当对指针 p 进行运算时, p_loc 也随之改变。

定义 3 表示指针 p 所指向堆内存的状态变量 $p_mem, p_mem \in \{0, 1\}$ 。 $p_mem = 0$ 表示该堆内存未用指针 p 释放, $p_mem = 1$ 表示该堆内存用指针 p 释放。

定义 4 二元组 (p_loc, p_mem) 记录指针 p 属性。则指针 p 上的更新操作可以定义为 (p_loc, p_mem) 上的相应变化。

规则 1 内存分配操作规则。C++ 程序设计语言提供了几个分配内存的函数,如 `strdup`、`malloc`、`calloc`、`realloc` 等,这里用 `malloc` 表示。如果 $p = \text{malloc}()$,则 $p_loc = 0, p_mem = 0$ 。

规则 2 指针赋值操作规则。如果 $q = p$,则 $q_loc = p_loc, q_mem = 0$ 。

规则 3 指针运算操作规则。如果 $p++$,则 p_loc++ , p_mem 不变;如果 $p--$,则 p_loc-- , p_mem 不变,其中 $n \in N$ 。

规则 4 内存释放操作规则。如果 `free(p)`,则 $p_mem = 1, p_loc$ 不变。

3.2 建立漏洞模型

通过对堆内存泄露漏洞类型的分析,内存泄露的一种情况是没有安全地释放(包括重复释放和部分释放)动态申请的堆内存。另一种情况是在函数返回时,局部指针变量的生命周期在函数返回后终止,从而导致内存泄露的发生。相应地,在使用堆内存释放操作 `free` 等、函数返回语句 `return` 之前对指针的属性进行约束,并且将这一约束描述为指针属性的条件表达式。如果代码不能满足这一约束条件,则可以确定为存在漏洞。一个程序中所有约束条件的总

和称为这个程序的内存泄露漏洞模型.

约束 1 内存释放操作 $\text{free}(p)$ 之前的约束条件:

$$(p_loc=0 \wedge p_mem=0 \wedge q_mem=0)$$

该约束条件表示在用 p 释放堆内存之前要确保 p 仍指向对内存首地址, 该堆内存未用指针 p 或 q 释放过.

约束 2 函数返回操作 return 之前的约束条件:

$$(p_mem! = q_mem)$$

该约束条件表示在函数返回之前要确保堆内存已经用 p 或 q 释放过.

3.3 漏洞模型转化为约束断言

将内存泄露漏洞模型的约束条件表达式转化为约束断言, 例如:

```
assert( $q\_loc == 0 \&\& p\_loc == 0 \&\& q\_mem == 0$ ); assert( $q\_mem != p\_mem$ )
```

内存泄露漏洞相关指针属性的更新操作和约束断言作为插入代码将插入到源代码中, 以实现断言可达性的检测.

4 静态检测系统

文中提出的内存泄露故障的静态检测系统的流程主要包括函数静态信息提取模块、程序插桩模块、模型检测模块等阶段, 如图 1 所示. 首先将 C 源代码输入函数静态语义信息提取模块, 得到包含所有语句语义信息的文件. 然后将该文件输入插桩模块, 得到插桩后的源代码. 最后将该源代码输入基于 Blast 的模型检测器, 得到安全漏洞检测结果.

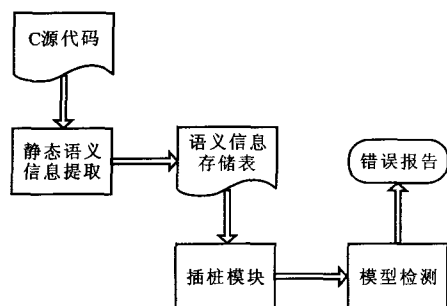


图 1 检测系统工作流程图

4.1 静态语义信息提取

在本系统中, 采用 Lex 与 Yacc 对 C 源程序进行静态分析. WBoxTool^[8] 工具是西安交通大学软件学院自主开发的针对 C 语言源程序的自动化白盒测试工具. WBoxTool 工具的前端—函数信息提取模块, 使用了 Lex 与 Yacc 完成对 C 源程序的词法、语

法分析和静态信息提取入库. 函数信息提取模块独立于其他模块, 可被后续模块使用^[9]. 文中提出的测试方法流程的函数静态语义信息的提取将直接使用 WBoxTool 工具函数信息提取模块, 将分析所得静态信息存入 PostgreSQL 数据库的语义信息存储表.

4.2 源代码静态插桩

插桩模块主要包括静态信息初始化、插桩类型判断、静态插桩等阶段, 完成对静态信息提取后的源程序, 按照插桩规则进行插桩, 实现将与内存泄露漏洞相关指针属性的更新操作和约束断言作为桩规则插入到源代码中. 下面给出了一个实例程序来说明插桩过程.

(1) 程序代码如下:

```
int main ( )
{
    int * p, * q, i, j=0;
    p=(int *) malloc (sizeof (int) * 5);
    for (i=0; i<3; i++)
    {
        if (j=0)
            p = p++ ;
    }
    q = p ;
    q -- = 3;
    free (q);
    q = NULL;
    free (p);
    p = NULL;
    return 0;
}
```

(2) 插桩后的代码如下:

```
#include <assert.h>
int main ( )
{
    int * p, * q, i, j=0;
    p = (int *) malloc (sizeof (int) * 5);
    p_loc=0;
    p_mem=0;
    for (i=0; i<3; i++)
    {
        if (j=0)
            p = p++ ;
            p_loc++
    }
    q = p;
    q_loc=p_loc;
```

```
p_mem=0;
q=q-3;
q_loc=q_loc-3;
assert (q_loc=0&&q_mem=0&& p_mem=0);
free (q);
q_mem=1;
q=NULL;
assert (p_loc=0&&p_mem=0&& q_mem=0);
free (p);
p_mem=1;
p=NULL;
assert (q_mem != p_mem);
return 0;
}
```

4.3 模型检测

Blast 是 UC Berkeley 开发的 C 语言程序验证工具,可以进行“EROOR”标签可达性检测、断言检测和时序安全检测等.断言检测是使用 Blast 最有效的方法之一.通过在 C 源程序中添加断言 assert(e),其中“e”代表一个 C 表达式,Blast 可以检测出断言中的表达式的值是否为零.本阶段将插桩后的 C 源代码输入 Blast 中,检测代码中的 Assert 断言是否正确,若不正确说明存在内存泄露漏洞.

5 实例测试

将本方法运用到实际中,设计了一个 157 行 C 源代码测试用例.在此测试用例中,设置 11 个内存泄露漏洞.测试的结果如表 1 所示.

表 1 测试结果分析

测试用例 代码行数	设置内存 泄露漏洞个数	检测出错误 路径个数	准确率
157	11	12	91%

从测试分析结果可以看出检测出的错误路径个数是已知漏洞个数的超集.

6 结束语

文中针对现有内存泄露漏洞检测方法的不足,提出了一种结合静态分析技术、源代码插桩技术和模型检测技术的静态检测方法.与已有技术相比,该

方法充分利用了现有模型检测工具对控制流和信息流的分析能力,将内存泄露的检测转化为插桩进源代码的约束断言的可达性判定问题.经实例证明了本方法的有效性和精确性.

虽然该方法在 157 行的实例程序检测上取得了较好的结果,但离大规模应用还有较大距离.将来会进一步完善内存泄露的漏洞模型,并提高检测的自动化水平.

参考文献:

[1] Nicholas Nethercote, Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation[C]// Proceedings of PLDI 2007. San Diego, California, USA, 2007.

[2] Heine D L. Static memory leak detection [D]. Stanford University, Department of Electrical Engineering, 2004.

[3] David L Heine, Monica S Lam. Static detection of leaks in polymorphic containers[C]// Proceeding of the 28th international conference on Software engineering. Shanghai, China, 2006.

[4] Clarke E M, Grumberg O, Peled D. Model checking[M]. Cambridge, UK: MIT Press, 2000.

[5] Henzinger TA, Jhala R, Majumdar R, et al. Lazy abstraction[J]. ACM SIGPLAN Notices, 2002, 37(1): 58-70.

[6] Chaki S, Clarke EM, Groce A, et al. Modular verification of software components in C[J]. IEEE Trans. on Software Engineering, 2004, 30(6): 388-402.

[7] William Landi. Undecidability of static analysis[J]. ACM Lett. Programming Languages Syst, 1992, 1(12): 323-337.

[8] 刘晓洪, 朱利. 白盒测试工具函数信息提取模块设计与实现[J]. 微电子学与计算机, 2005, 22(5): 166-169.

[9] 高海昌, 朱利. 面向上下文无关语言函数信息提取模块的设计[J]. 计算机工程, 2006, 32(4): 49-51.

作者简介:

付晓毓 女, (1984-), 硕士研究生. 研究方向为软件安全性、软件测试.

朱 利 男, (1967-), 博士, 副教授. 研究方向为软件工程、网络技术与多媒体通信等.

顾 伟 男, (1982-), 硕士研究生. 研究方向为软件工程.