基于 CFG 的函数调用关系静态分析方法①

黄双玲, 黄章进, 顾乃杰

(中国科学技术大学 计算机科学与技术学院, 合肥 230027)
(中国科学技术大学 安徽省计算与通信软件重点实验室, 合肥 230027)
(中国科学技术大学 先进技术研究院, 合肥 230027)

摘 要: 函数调用关系能够反映软件系统中函数间的依赖关系, 完整的函数调用关系可以更好地辅助程序验证和死锁分析, 提升验证和分析的完备性. 现有静态分析函数调用关系的方法不能准确分析函数指针和虚函数的调用, 影响了其分析结果的准确性. 针对这一问题本文提出了一种基于控制流图(Control Flow Graph, CFG)的函数调用关系静态分析方法, 该方法首先使用 GCC 插件静态获取源代码中的类型和函数 CFG 等信息并构建分析路径, 然后采用本文提出的模拟仿真算法分析程序中的语句, 并解析函数指针和虚函数的调用, 最后基于分析结果生成完整的函数调用关系. 实验结果表明, 该方法能够很好地支持对函数指针和虚函数的处理, 提升了分析结果的准确性.

关键词: 函数调用关系; 静态分析; 控制流图; 函数指针; 虚函数

Static Analysis Method of Generating Function Call Relations Based on CFG

HUANG Shuang-Ling, HUANG Zhang-Jin, GU Nai-Jie

(School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China)
(Anhui Province Key Laboratory of Computing and Communication Software, Hefei 230027, China)
(Institute of Advanced Technology, University of Science and Technology of China, Hefei 230027, China)

Abstract: Function call relations can be used to reveal the dependency relations between functions in software systems. An integrated function call relation makes a good contribution to program verification and deadlock analysis, and improves the completeness of verification and analysis. Existing methods of function call relations based on static analysis do not provide integrated support for function pointers or virtual functions, which lowers the accuracy of analysis result. This paper proposes a static analysis method for function call relations generation based on Control Flow Graph (CFG). This method gains source file information by a GCC plugin, and predicts function analysis paths. Then it uses a simulation algorithm proposed by this paper to analyze the statements of those paths, while the call of function pointers and virtual functions are analyzed accurately. It generates full function call relations based on the analysis result. Experimental results show that this method can support the analysis of function pointers and virtual functions, and improves the accuracy of analysis results.

Key words: function call relations; static analysis; control flow graph; function pointer; virtual function

1 引言

函数调用关系能够反映软件系统中函数间的依赖关系, 在程序的理解与分析、软件的测试与维护、编译优化、过程间数据流分析、回归测试等众多软件工程领域中都有着广泛的应用. 完整的函数调用关系可以更好地辅助程序验证和死锁分析, 提升验证和分析的完备性.

函数调用关系的分析方法包括静态分析方法和动态分析方法. 静态分析方法是指在不运行待分析程序的前提下进行调用关系分析. 在大型 C/C++工程项目

① 基金项目:安徽省自然科学基金(1408085MKL06);高等学校学科创新引智计划资助(B07033)
收稿时间:2015-03-10;收到修改稿时间:2015-04-26

中, 函数指针和虚函数被广泛使用, 然而现有静态分析函数调用关系的方法不能准确分析此类复杂语法, 这给程序分析带来了一定的困难. CG-RTL[1]根据 Eygpt[2]的工作原理, 利用编译器输出的寄存器传送语言(Register Transfer Language, RTL)中间结果, 提取目标代码的函数调用关系, 但是没有分析利用函数指针实现的函数调用. RacerX[3]对函数指针只做了简单处理, 记录对同类型函数指针赋值的所有函数, 在函数指针调用时, 假定这些函数均会被调用, 这种处理方法必然会产生一些不必要的函数调用, 导致函数分析的不准确, 而且没有处理虚函数调用. Bacon 等[4]采用 RTA 方法分析了虚函数的调用, 通过源码中的类层次关系和上下文中已创建的类的对象来分析虚函数, 但是当程序同时创建了基类和派生类的对象时, 依据已有的对象则无法确定虚函数对应的函数. 与静态分析方法相对应地动态分析方法则是在程序运行时记录函数调用的实际情况, 可以解决静态分析方法中获取信息不准确的问题, 但是受程序输入的限制, 不能得到所有的调用路径, 逻辑上不具有完备性.

本文提出了一种基于 CFG 的函数调用关系静态分析方法, 与已有静态分析方法相比, 可以更为准确地分析函数指针和虚函数调用. 本方法的主要流程如下: 首先利用 GCC 插件静态获取源码中的类型和函数 CFG 等信息, 建立信息的结构模型并构建函数分析路径; 然后采用本文提出的模拟仿真算法静态分析程序中的语句, 并解析函数指针和虚函数的调用; 最后基于分析结果生成完整的函数调用关系. 实验结果表明, 该方法能够很好地支持对函数指针和虚函数的分析, 提升了分析结果的准确性.

本文剩余部分组织如下: 第 2 章介绍背景, 第 3 章介绍函数调用关系的静态分析方法, 第 4 章通过实验给出具体实例分析, 第 5 章给出总结.

2 背景

2.1 信息提取

信息提取是静态分析的基础. 现有信息提取技术中, 广泛应用于程序语义分析的 GCC 抽象语法树[5] (Abstract Syntax Tree, AST)文件虽能够包含整个编译单元的完整表示, 记录了源程序中的语法结构, 但不方便程序分析. 由于 GCC 编译源程序时产生的 CFG 文件最接近源程序本身的语义, 本文利用 GCC 提供的插件功能, 将 CFG 文件的信息以格式化的形式输入到文件中, 以便于信息提取.

2.2 GCC 插件

插件是 GCC4.5 版本后推出的可改变 GCC 行为的动态链接库文件, 它提供可注册的事件列表和事件对应的回调函数接口. 插件提供的事件贯穿于整个 GCC 编译的过程, 包括词法分析、语法分析和中间代码优化等阶段, 如语法分析时插件提供 PLUGIN_FINISH-_DECL 事件操作程序声明的信息[6].

本文实现了一个 GCC 插件用于静态提取源文件中的类型和函数 CFG 等信息. 该插件根据 GCC 提供的事件功能, 注册用于获得源文件中类型定义和函数信息的回调函数. 在 GCC 执行完对函数 CFG 的优化后, 立即调用获取函数信息的回调函数, 函数信息包括函数的返回值、参数列表、局部变量列表、基本块[7]和 CFG 等信息. 由于程序分析中涉及到函数指针和虚函数, 还需要获取程序中的类型信息. 在 GCC 解析完文件中的类型后, 调用获取文件中类型定义的回调函数, 类型信息包含程序中的命名空间、类、结构体和函数指针等.

3 函数调用关系的静态分析方法

本文提出了一种基于 CFG 的函数调用关系的静态分析方法, 其框架图如图 1 所示. 前端首先在源代码的编译过程中加载自写的 GCC 插件, 获取源文件中的类型和函数 CFG 等信息, 对 C++程序还需额外获取类的虚函数表信息; 然后根据获取的文件信息建立类型和函数的结构模型并构建分析路径, 展开函数体内的分支. 后端采用本文提出的模拟仿真算法模拟程序的执行过程, 并解析函数指针和虚函数的调用, 最后生成完整的函数调用关系.

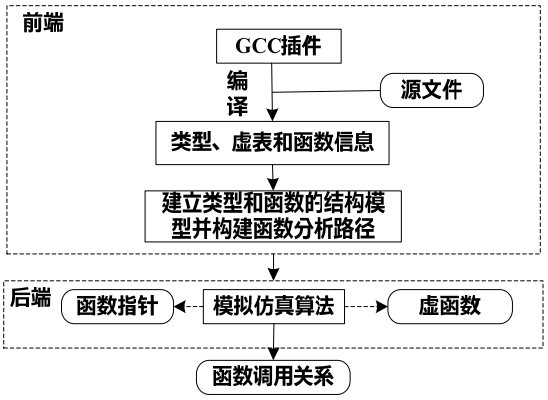


图 1　静态分析方法的框架图

3.1 信息结构模型的建立

在源码编译过程中, 通过加载 2.2 节中实现的 GCC 插件, 获取源文件中的类型和函数信息以及虚函数表信息.

类型模型中存储了类的成员列表、继承关系和虚函数表信息, 用于封装程序中类型的定义、声明、初始化和赋值等操作. 函数模型中存储函数所在文件、返回值、参数列表、局部变量列表、CFG 和分析路径等信息. 函数的分析路径依据 CFG 的结构, 采用文献[8]提出的改进 DFS 算法来构建. 为了解决静态函数和函数重载导致的函数同名问题, 本文依据函数所在文件、函数名和参数类型来唯一确定函数.

3.2 函数指针

本文在程序分析过程中解析并记录与函数指针相关变量的值, 当分析到函数指针的调用时, 通过查找记录获取其指向的实际函数. 由于函数指针在 C/C++程序中的表现形式不同, 下面针对 C 和 C++中的函数指针分别作分析.

3.2.1 C 程序中的函数指针

C 程序中函数指针变量作为单独的变量或者结构体的成员变量存在, 解析函数指针的难点在于函数指针可能存在于复杂的嵌套结构体中. CFG 中与函数指针相关的赋值包括函数指针的直接赋值、结构体的成员赋值、取地址和嵌套赋值四种.

本文实现了一个通用的赋值方法, 用于支持上述四种类型的赋值. 为了记录指针变量和结构体变量的值, 需要在类型模型中记录存储变量已赋值成员列表和成员对应的值列表.

```
1.struct A {  void (*pFunc)();  };     8.    bb.c = 1;
2.struct B {  struct A aa;  int c; };  9.    bb.aa.pFunc = funcA;
3.void test_func(struct A *para) {     10.   if (bb.c)
4.    para->pFunc = funcB;             11.       test_func(&bb.aa);
5.}                                    12.   bb.aa.pFunc();
6.void main() {                        13.}
7.    struct B bb;
```

图 2 C 程序中的函数指针

以图 2 中的程序为例, 函数指针作为结构体的成员变量存在, 假设 funcA 和 funcB 为已知的两个函数, main 函数有两条分析路径, 分别是 main->test_func ->bb.aa.pFunc 和 main->bb.aa.pFunc, 两条分析路径上均有函数指针的调用. 在分析过程中, main 函数中的每条分析路径均会分析第 9 行的语句, 即对嵌套结构体的成员赋值, 对于此类复杂的结构体嵌套赋值, 首先将其划分为非嵌套结构体成员赋值和结构体赋值, 然后对划分后的语句分别处理. 按照我们的划分规则, 第 9 行的赋值语句被划分为 tmp_aa.pFunc = funcA 和 bb.aa = tmp_aa 两条赋值语句, tmp_aa 是和 aa 同类型的临时变量. 上述划分与原有的嵌套结构体赋值是等价的, 因为第一条赋值语句保证了 tmp_aa 的成员变量 pFunc 的值为赋值语句的右值 funcA; 在处理第二条赋值语句时, 会将 tmp_aa 中已赋值的成员 pFunc 的值拷贝一份到 bb 的成员变量 aa. main 函数的第一条分析路径会分析到第 11 行的函数调用语句, 当分析函数 test_func 时, 首先将实参的值赋值给形参, 即 bb.aa 赋值给 test_func 函数的形参 para. 然后分析 test_func 函数内的赋值语句(第 4 行), 对形参 para 的值进行修改. 由于该函数的参数传递方式为取地址赋值, 因而 para 和 bb.aa 的值指向类型模型中成员变量值列表的同一块内存, 这样 bb.aa 的值也被修改了. 当函数 test_func 调用返回时, 继续分析下一条语句(第 12 行), 即函数指针的调用, 此时首先查询函数指针变量的值, 获取指针指向的函数为 funcB, 从而得知第一条分析路径为 main->test_func->funcB. 对于 main 函数的第二条分析路径, 首先分析第 9 行的赋值, 然后分析第 12 行的函数指针调用, 此时查询函数指针的值为 funcA, 所以第二条分析路径为 main->funcA.

3.2.2 C++程序中的函数指针

C 程序中针对函数指针的处理可以扩展到 C++程序中, 但在 C++程序中存在一种特殊的函数指针, 即指向类成员函数的指针. 此类指针分为两种情况, 一种是指向类的非虚成员函数指针, 另一种是指向类的虚成员函数指针. 处理这两种函数指针需要依据 GCC 对类继承的内存分布机制. 在函数的 CFG 中, 通过一个结构体类型来描述指向类成员函数的指针信息, 该结构体中有两个成员变量__pfn 和__delta.

以图 3 中 16 行指向类 Child 的成员函数的指针 pfun 为例, 其在 CFG 中的表现形式为:

```
struct {
    int Child::<T3cd6>(struct Child*) *__pfn;
    long int __delta;
}pfun;
```

pfun.__pfn 的值为指针实际指向的函数, pfun.__delta

的值为指针指向的类(Child)和定义目标函数的类在内存中的偏移. 图 4 为 Child 类的内存分布图, Base_1 和 Base_2 类分别有自己的虚函数表, vptr 为指向虚函数表的指针. Child 类继承了 Base_1 和 Base_2 以及它们各自的虚表, Child 类的虚成员函数在第一个父类 Base_1 的虚表中, 并重写了父类的虚函数 foo_1.

```
1 .class Base_1 {
2.    int a;
3. public: virtual int foo_1() { return 1;}
4.          virtual int foo_2() { return 2;}
5. };
6. class Base_2 {
7.    int b;
8. public: virtual int foo_3() { return 3;}
9. };
10.class Child：public Base_1,Base_2{
11.    int c;
12.public: virtual int foo_1() { return 4;}
13.};
14.int main() {
15.  Child *child = new Child();
16.  int (Child::*pfun)();
17.  pfun = &Child::foo_2;
18.  int res1 = (child.*pfun)();
19.  pfun = &Child::foo_3;
20.  int res2 = (child.*pfun)();
21.  Base_1 *obj = child;
22.  int res3 = obj->foo_1();
23.}
```

图 3　指向类成员函数的指针和虚函数

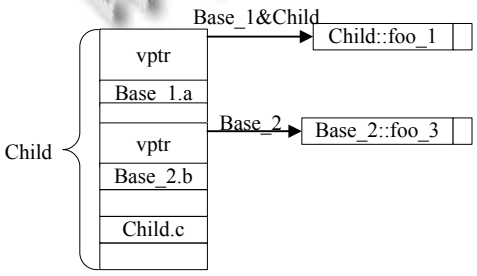


图 4　Child 类对象的内存分布图

当指针指向类的非虚成员函数(17 行)时, pfun.__pfn 的值为函数名(Base_1::foo_2), pfun.__delta 为 0. 当指针指向类的虚成员函数(19 行)时, pfun.__pfn 的值形式为“sizeof(int *)*N+1”, 表示指针指向目标类的虚表中第 N 个虚函数. 在此例的 CFG 中, pfun.__delta 的值为 16, 可计算出目标类为 Base_2; pfun.__pfn 的值为“(8*N+1) = 1”(64 位系统中), 表示类 Base_2 虚表中的第一个虚函数即 Base_2::foo_3.

3.3 虚函数

C++程序中虚函数的处理是程序分析中的一个重要问题, 本文对此也做了相关研究. 在函数的 CFG 中, 会为虚函数调用提供指针变量 obj 和指针指向的虚函数在目标类对应的虚函数表中的偏移 num, 表现形式为“OBJ_TYPE_REF(var,obj ->num)”.

图 3 中关于虚函数调用在 CFG 中的表现形式如图 5 所示. 在分析虚函数调用前, 首先需要知道指针变量 obj 的实际类型, 为此本文在存储变量的结构模型中记录了变量的定义类型 varType 和变量指向内存空间的实际类型 realType. 图 3 的程序中, child 定义类型为 Child, 实际类型通过图 5 中的调用语句(第 2 行)Child 的构造函数来确定, 即 Child 类型. 变量 obj 的定义类型为 Base_1, 实际类型通过赋值语句(第 7 行)来记录, 该赋值语句属于类型转换, 可以确定 obj 的实际类型为 Child.

```
1. BB_STMT_BEGIN:
2.    TYPE:GIMPLE_CALL
3.    LHS:
4.    NAME:Child::Child
5.    PARA:type:void*      name:child
6. BB_STMT_BEGIN:
7.    TYPE:GIMPLE_ASSIGN
8.    LHS:obj              RHS:child
9. BB_STMT_BEGIN:
10.   TYPE:GIMPLE_CALL
11.   LHS:
12.   NAME:OBJ_TYPE_REF(D.29;obj->0)
13.   PARA:type:Base_1*  name:obj
```

图 5　虚函数调用在 CFG 中的形式

分析虚函数调用语句(第 10 行)时, 首先根据 11 行提取指针变量名 obj 和变量实际类型在虚函数表中的偏移 num; 然后在类型记录中查询 obj 的真实类型, 并找到其对应的虚函数表; 最后根据 num 的值查询虚函数表, 准确找到对应的虚函数. 本例中指针变量 obj 的实际类型为 Child, 偏移 num 为 0, 实际调用的函数为 Child 的第 1 个虚函数, 即 Child::foo_1.

3.4 模拟仿真算法

为了准确解析函数指针和虚函数的调用, 在程序分析中, 本文提出了一个可以模拟程序动态执行过程的仿真算法. 该算法依据函数的分析路径, 逐条分析该路径上的语句, 解析函数指针和虚函数的调用, 并记录函数调用信息. 为了完整覆盖程序的所有可能执行路径, 本文首先分析每个被调函数的一条分析路径, 当程序的一条完整路径分析完后, 回溯分析该条路径上被调函数的其他待分析路径.

仿真算法的具体实现如算法 1 所示, 算法的输入是程序的入口函数(main 函数), 输出是完整的函数调用关系. 算法的执行流程如下: 首先将当前分析的函数初始化为入口函数(第 1-2 行), 取出入口函数的一条分析路径 path; 然后逐条分析 path 上的语句 stmt(第 5 行), 根据语句的类型作相应的处理(第 7-9 行). 当一条

路径分析完后回溯分析该条路径上被调函数的其他待分析路径(第 11 行); 最后将 path 更新为入口函数的下一条待分析路径继续分析(第 12 行).

算法 1. 模拟仿真算法

输入: 程序的入口函数: entry_node

输出: 完整的函数调用关系

```
function_analysis(entry_node)
1.    cur_func = entry_node;
2.    cur_func->prev = cur_func->next = NULL;
3.    path = entry_node->path;
4.    while(path)
5.        for each stmt in path do
6.            switch(stmt->type)
7.                case '赋值': 赋值处理;break;
8.                case '调用': 函数调用处理; break;
9.                case '返回': 函数返回; break;
10.           end case
11.       backtrace();//回溯分析
12.       path = path->next;
13.   end while
```

仿真算法的核心部分为对操作栈和函数调用链的维护以及函数指针和虚函数的处理. 操作栈中存储程序分析中已处理的语句信息, 用于回溯分析. 函数调用链中记录已分析函数的调用关系. 程序分析时, 逐条分析一条路径上的语句, 并将已处理的语句存入操作栈中. 处理的语句类型包括赋值语句、函数调用语句和函数返回语句. 当遇到赋值语句时, 采用 3.2.1 节提出的通用赋值方法, 将变量的值存储在类型模型中; 遇到调用返回语句时, 更新当前分析的函数为上层调用函数, 并继续分析其下一条语句.

对函数调用语句的处理, 具体流程如图 6 所示, 首先判断调用语句的函数名是否为函数指针变量, 如果是, 则查询指针变量的值, 以确定当前实际调用的函数; 否则判断是否为虚函数调用, 若是, 则根据指针指向的实际类型和函数在虚函数表中的偏移获取调用的函数名, 否则调用语句的函数名即为实际调用的函数. 最后根据函数名和调用语句的参数类型, 查找并分析被调函数. 为了提高分析效率, 避免同一函数的重复分析, 在处理被调函数时, 需要判断其是否已被分析过. 若已被分析过, 则进行上下文敏感的别名分析[9]; 否则分析被调函数. 分析被调函数时, 首先将实参赋值给被调函数的形参, 然后再分析被调函数分析路径上的语句, 以实现函数间调用的信息传递, 并在指针或引用作为参数时修改上层调用函数实参的值.

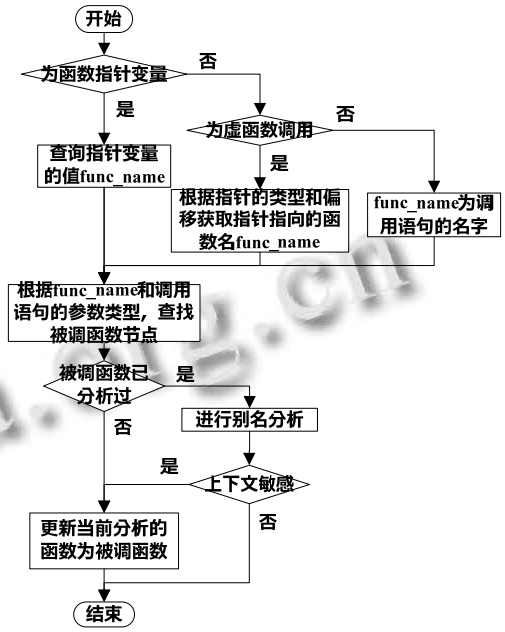


图 6 处理调用语句的流程图

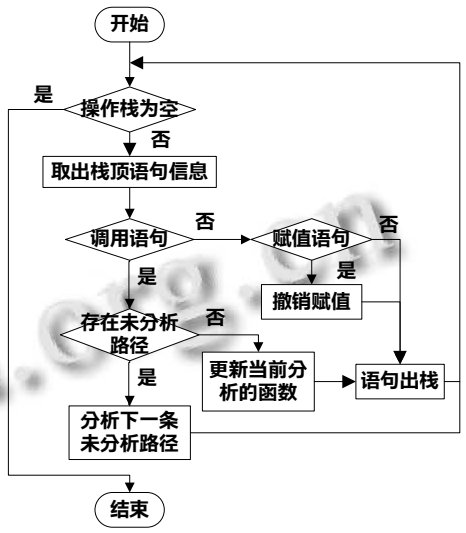


图 7 回溯分析过程的流程图

回溯分析过程依赖于程序分析中维护的操作栈, 具体流程如图 7 所示. 首先获取栈顶的语句和当前待分析的函数信息, 然后根据栈顶语句的类型作相应处理. 若语句类型为函数调用语句, 判断被调函数是否存在未分析路径, 若存在则将其作为当前分析路径; 否则说明该函数的所有路径已分析完, 更新当前分析的函数为函数调用链中的上一个分析函数, 并将该语句信息出栈. 若语句类型为赋值语句, 撤销先前的赋

值操作, 使算法回到赋值前的状态并将此语句出栈. 若语句类型为函数返回语句, 直接将此语句出栈.

4 实验与分析

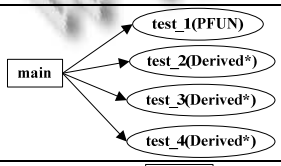
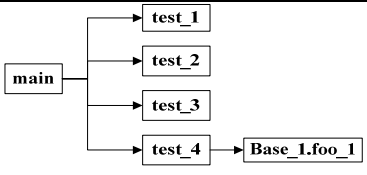
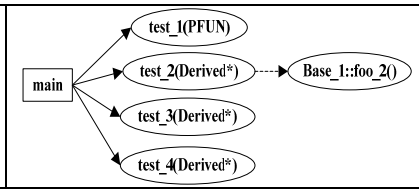
本文在静态分析 C/C++程序函数调用关系中实现了对函数指针和虚函数的解析, 通过与现有静态分析工具 CodeViz[10]、SourceInsight[11]和 Egypt 的对比来验证本方法的正确性. 如图 8 中的程序, 假设普通函数 funA 已知, 函数 test_1、test_2、test_3、test_4 分别为 C 程序中的函数指针、C++中指向类非虚成员函数、指向虚成员函数和虚函数调用. 本文利用 Graphviz[12] 工具绘制了程序的仿真结果图, 表 1 是本文的仿真算法与现有工具的分析结果对比.

```
1. typedef void (*PFUN)();
2. class Base_1 {
3. public: virtual void foo_1() { return;}
4.         void foo_2() {return;}
5.};
6. class Base_2 {
7. public: virtual void foo_3() { return;}
8.};
9. class Derived: public Base_1,Base_2{
10. public: virtual void foo_1() { return;}
11.};
12.typedef void (Derived::*CPFUN)();
13.void test_1(PFUN pfun) {
14.   pfun();
15.}
16.void test_2(Derived *d) {
17.   CPFUN cpf = &Derived::foo_2;
18.   (d->*cpf)();
19.}
20.void test_3(Derived *d) {
21.   CPFUN cpf = &Derived::foo_3;
22.   (d->*cpf)();
23.}
24.void test_4(Derived *d) {
25.   Base_1 *b = d;
26.   b->foo_1();
27.}
28.void main() {
29.   Derived *d = new Derived();
30.   int a = 2;
31.   if (a == 1)
32.      test_1(funA);
33.   else if (a == 2)
34.      test_2(d);
35.   else if (a == 3)
36.      test_3(d);
37.   else
38.      test_4(d);
39.}
```

图 8　函数指针和虚函数的综合实例

表 1 中, CodeViz 对函数指针和虚函数调用均不能做出分析; SourceInsight 不能解析函数指针, 虽然解析出了虚函数, 但分析结果不正确; Egypt 不能解析虚函数, 对于函数指针只支持 C++中指向类的非虚成员函数.

表 1　仿真算法与现有分析工具的分析结果

工具名	分析结果
CodeViz	
Source-Insight	
Egypt	

从本文提出的仿真算法分析结果可以看出, 该算法准确地分析了函数指针和虚函数的调用, 并给出了函数的参数类型(以'^'分隔).

5 总结

本文针对现有静态分析方法不能准确分析函数指针和虚函数的不足, 提出了一种基于 CFG 的函数调用关系静态分析方法. 实验结果表明, 本文实现的静态分析方法可以很好地支持 C/C++程序中函数指针和虚函数的处理, 提升了分析结果的准确性. 相比现有工具, 该方法生成的函数调用关系准确性更高, 在静态死锁检测和路径覆盖测试等研究领域中均可以得到推广使用.

参考文献

1 孙卫真,杜香燕,向勇,等.基于 RTL 的函数调用图生成工具 CG-RTL.小型微型计算机系统,2014,35(3):555–559.

2 Gustafsson Andreas. Egypt: create call graph from GCC RTL dump. http://www.gson.org/egypt/egypt.html. [2013-06-23].

3 Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. Operating Systems Review, 2003, 37(5): 237–252.

4 Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. ACM Sigplan Notices, 1996, 31(10): 324–341.

5 林立,王毅刚,叶飞.基于 GCC 的 C/C++源程序静态信息提取技术.计算机与数字工程,2011,39(2):152–155.

6 Stanley DM, Xu D, Spafford EH. Improved kernel security through memory layout randomization. 2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC). IEEE. 2013. 1–10.

7 Zheng Y, Mu Y, Zhang Z. Research on the static function call path generating automatically. The 2nd IEEE International Conference. 2010. 405–409.

8 Joseph P. A method to determine a basis set of paths to perform program testing. http://hissa.nist.gov/publications/nistir5737/#section1.

9 张立勇.软件源代码安全分析研究[学位论文].西安:西安电子科技大学,2011.

10 Mgebrova K. CodeViz: a callgraph visualizer. http://www.csn.ul.ie/~mel/projects/codeviz. [2012-02-16].

11 Source Insight: need to understand code. http://www.sourceinsight.com/support.HTML#Docs. [2013-03-16].

12 Ellson J, Gansner E, Koutsofios L, et al. Graphviz-open source graph drawing tools. Graph Drawing. Springer Berlin Heidelberg, 2002: 483–484.