第41卷 第6A期
2014年6月

计 算 机 科 学
Computer Science

Vol. 41 No. 6A
June 2014

C语言静态代码分析中的调用关系提取方法

江梦涛 荆 琦
(北京大学软件与微电子学院 北京 100871)

摘 要 程序静态分析(Program Static Analysis)是指在不运行代码的方式下,通过词法分析、语法分析、控制流分析等技术对程序代码进行扫描,验证代码是否满足规范性、安全性、可靠性、可维护性等指标的一种代码分析技术。首先对程序静态分析的特点、常用静态分析技术、静态分析实现方式进行描述,然后通过一个实例讲解C语言静态代码分析中函数与变量的调用关系的实现方法,总结了现今在C语言调用关系分析中可以借鉴的工具与实现方式。

关键词 程序静态分析,函数调用,代码安全
中图法分类号 TP314 **文献标识码** A

Method of Extracting Function Call Relationship in Static Code Analysis of C Language

JIANG Meng-tao JING Qi
(School of Software and Microelectronics, Peking University, Beijing 100871, China)

Abstract Program Static Analysis is a method, of verificating whether code could meet the specification, safety, reliability, maintainability and other indicators with scanning the program code through lexical analysis, syntax analysis and control flow analysis techno-logy, and without running the code actually. This paper presents the characteristics, technologies, implementations of static code analysis, and then explain the methods of parsing function calls in C language static code analysis.

Keywords Static code analysis, Function call, Code security

1 引言

程序静态分析(Program Static Analysis)是指在不运行代码的方式下,通过词法分析、语法分析、控制流分析等技术对程序代码进行扫描,验证代码是否满足规范性、安全性、可靠性、可维护性等指标的一种代码分析技术[1]。

程序静态分析可以帮助软件开发人员、质量保证人员查找代码中存在的结构性错误、安全漏洞等问题,从而保证软件的整体质量。

本文首先对程序静态分析的特点、常用静态分析技术、静态分析实现方式进行描述,然后通过几个实例讲解C语言静态代码分析中函数与变量的调用关系的实现方法,总结了现今在C语言调用关系分析中可以借鉴的工具与实现方式。

2 静态代码分析的特点

程序静态分析是与程序动态分析相对应的代码分析技术,它通过对代码的自动扫描发现隐含的程序问题,主要具有以下特点:

(1)不实际执行程序。动态分析是通过在真实或模拟环境中执行程序进行分析的方法,多用于性能测试、功能测试、内存泄漏测试等方面。与之相反,静态分析不运行代码只是通过对代码的静态扫描来对程序进行分析。

(2)执行速度快、效率高。目前成熟的代码静态分析工具每秒可扫描上万行代码,相对于动态分析,具有检测速度快、效率高的特点。

(3)误报率较高。代码静态分析是通过对程序扫描找到匹配某种规则模式的代码从而发现代码中存在的问题,例如可以定位strcpy()这样可能存在漏洞的函数,这样有时会将一些正确代码定位为缺陷,因此静态分析有时存在误报率较高的缺陷,可结合动态分析方法进行修正。

3 常用静态分析技术

常用的静态代码分析技术如下:

(1)词法分析:从左至右将字符逐个读入源程序,对构成源程序的字符流进行扫描,通过使用正则表达式匹配方法将源代码转换为等价的符号(Token)流,生成相关符号列表,Lex为常用分析工具。

(2)语法分析:判断源程序结构上是否正确,通过使用上下文无关语法将相关符号整理为语法树,Yacc为常用工具。

(3)抽象语法树分析:将程序组织成树形结构,树中相关节点代表了程序中的相关代码,目前已有Javacc等抽象语法树生成工具。

(4)语义分析:对结构上正确的源程序进行上下文有关性质的审查。

(5)控制流分析:生成有向控制流图,用节点表示基本代码块,节点间的有向边代表控制流路径,反向边表示可能存在

本文受"核高基"科技重大专项:操作系统内核分析和安全性评估(2012ZX01039-004)资助。
江梦涛(1989—),男,硕士生,主要研究方向为开源操作系统,E-mail:insomniacdoll@gmail.com。

的循环，还可生成函数调用关系图，表示函数间的嵌套关系。

(6)数据流分析：对控制流图进行遍历，记录变量的初始化点和引用点，保存相关数据信息。

(7)污点分析：基于数据流图判断源代码中哪些变量可能受到攻击，是验证程序输入、识别代码表达缺陷的关键。

程序静态分析是在不执行程序的情况下对其进行分析的技术，简称为静态分析。而程序动态分析则是另外一种程序分析策略，需要实际执行程序。大多数情况下，静态分析的输入都是源程序代码，只有极少数情况会使用目标代码。静态分析这一术语一般用来形容自动化工具的分析，而人工分析则往往叫做程序理解。

静态分析越来越多地被应用到程序优化、软件错误检测等领域。比如，Coverity Inc. 的软件质量检测产品就是利用静态分析技术进行错误检测的成功代表。

静态程序分析的复杂程度依所使用的工具而异，简单的只考虑个别叙述及声明的行为，复杂的可以分析程序的完整源代码。不同静态程序分析产生的信息也有所不同，简单的可以是标示可能的代码错误(如 lint)，复杂的可以是形式化方法，也就是用数学的方式证明程序的某些行为符合其设计规格。

软件度量和反向工程可以视为一种静态程序分析的方式。实际应用中，在定义所谓的软件质量指针(software quality objectives)后，软件度量的推导及程序分析常一起进行，在开发嵌入式系统时常会用这种方式进行[2]。

静态程序分析的商业用途可以用来验证安全关键计算机系统中的软件，并指出可能有计算机安全隐患的代码，这类的应用越来越多[3]。

4 C 语言中的静态代码分析

C 语言的灵活性带来了代码效率的提升，但相应带来了代码编写的随意性，另外 C 编译器不进行强制类型检查，也带来了代码编写的隐患。识别并报告 C 语言中的编程陷阱和格式缺陷是静态代码分析中的难点。这些难点包括：需要对 C 语言程序进行全局分析，识别没有被适当检验的数组下标，报告未被初始化的变量，警告使用空指针、冗余的代码，等等。在程序动态测试之前发现编码错误，是 C 语言静态代码分析的目标。

本文将着重介绍现在在 C 语言静态代码分析过程中进行函数调用关系提取时可以采取的方法和可以运用的工具。

5 函数调用关系的理论基础

函数调用关系的数学表达如下：

函数调用流图可用于对被测软件做静态规约检查和质量评估。它是一个有向图，其形式化定义为 $FunCallGraph(V, R)$，V 是函数调用流图中所有节点的有穷非空集合，R 是节点间的有向边的有限集合，并满足 $R \subseteq V \times V$。具体定义如下：

$$V=\{x \mid x \in funSet\}$$

$$R=\{VR\}$$

$$VR=\{\langle x,y\rangle \mid P(x,y) \wedge (x \in V, y \in V)\}$$

其中，VR 是函数调用流图中节点之间的关系集合；$funSet$ 表示函数调用流图中的节点集合；谓词 $P(x,y)$ 表示 x 到 y 的一条单向通路。

6 函数调用关系的理论基础

6.1 使用正则表达式提取调用关系

使用正则表达式提取调用关系是最简单的办法。其流程大致是扫描项目中的每一个源码文件，匹配源码中所有函数定义，将其函数名、参数列表、所在文件等信息存储起来；然后对所有源码再进行第二次扫描，此次将会匹配每一个函数中调用的函数，再将这种调用关系存储起来即可。

图 1 所示的正则表达式可以匹配 C 语言中的函数定义。

```
(?:(extern|static)(?:\s+))?                          #修饰符
(?:                                                  #返回值类型
    (
        (?:[a-zA-Z_]\w*)                             #有效的符号
        (?:(?:\s(?:\s*\*\s*)*)|(?:(?:\s*\*\s*)*\s))  #指针类型符号
    )
    ([a-zA-Z_]\w*)\s*                                #函数名
)
(\([^\)]*\))\s*                                      #参数列表
(?:                                                  # {} 或者 ;
    (\{                                              #左大括号
        (?:[^\{\}]*)                                 #函数体
        (?:
            (?:(?'brace'\{)[^\{\}]*)+
            (?:(?'-brace'\})[^\{\}]*)+
        )*
        (?(brace)(?!))
    \})                                              #右大括号
    |;
)
```

图 1 匹配 C 语言中的函数定义的正则表达式

值得注意的是，上面的正则表达式是 C# 风格的，如果在其他编程语言中使用，需要一定的修改。

按照同样的方法，在识别出函数定义以后，可以使用正则表达式的分组功能，将其中的函数体提取出来，然后对函数体进行正则匹配，取得其中的函数调用。函数调用的正则匹配较为简单，这里不再赘述。

6.2 使用开源软件 ctags 和 cscope 进行调用关系的提取

ctags 是一个用于从程序源代码树产生索引文件(或 tag 文件)，从而便于文本编辑器来实现快速定位的实用工具。在产生的 tag 文件中，每一个 tag 的入口指向了一个编程语言的对象。这个对象可以是变量定义、函数、类或其他的物件。

ctags 是开放源代码的程序，支持下列编程语言：汇编，AWK，ASP，BETA，Bourne/Korn/Zsh Shell，C，C++，COBOL，Eiffel，Fortran，Java，Lisp，Lua，Make，Pascal，Perl，PHP，Python，REXX，Ruby，S-Lang，Scheme，Tcl，Vim，以及 YACC。

支持 ctags 产生的 tag 文件的编辑器以及编辑器插件包括：Vim、Vile、Lemmy 等等。

cscope 是一个 C 语言的浏览工具，通过这个工具可以很方便地找到某个函数或变量的定义位置、被调用的位置等信息，目前支持 C 和 C++。cscope 自身带一个基于文本的用户界面，不过 gvim 提供了 cscope 接口，因此可以在 gvim 中调用 cscope，方便快捷地浏览源代码。

从 ctags 中可以很方便地获取函数以及变量的依赖关系，下面是一个获取变量依赖关系的例子。

比如 ctags 的中间索引文件中有图 2 所示的一个片段。

图 2 ctags 的中间索引文件片段

其中每一行就是一个符号说明，第一列是符号的名称；第二列是符号所在的文件；第三列是找到这个符号所用的正则

表达式，第四列是这个符号的属性，比如 m 代表此符号是一个结构体之中的成员变量，f 代表是一个函数，v 代表普通变量，s 代表结构体声明，d 代表一个 define 宏定义。这里我们比较关心的是属性为 s 和 m 时，怎样提取变量之间的关系。当这个符号为 m 时，之后紧跟的一列就表示引用这个变量所声明的结构体名，如果这个成员变量本身还是一个结构体，再向后一列就是这个成员变量本身结构体的名字。通过对每一行的解析，可以获取到变量之间的依赖关系。

图 3 是使用 python 写的一个小例子的运行结果。

```
E:\WIN7\Dev\Workspaces\linux-3.5.4>python test.py
members of struct msgtemplate:
        file:
        buf
        typeref:struct:msgtemplate::genlmsghdr  g
        typeref:struct:msgtemplate::nlmsghdr    n
```

图 3 提取 ctags 的中间索引文件关系样例

可以看到，图 3 中的运行结果已经成功地将图 2 中高亮的关系表示出来了。

之后需要做的就是根据数据库表的设计将这个关系存储到数据库表中即可。

6.3 使用抽象语法树构建的方法进行调用关系的提取

抽象语法树是源代码的抽象语法结构的树状表现形式，这里特指编程语言的源代码。树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现，而类似于 if-condition-then 这样的条件跳转语句，可以使用带有两个分支的节点来表示。

抽象语法树的结构不依赖于源语言的文法，也就是语法分析阶段所采用的上下文无关文法。因为在 Parser 工程中，经常会对文法进行等价的转换（消除左递归、回溯、二义性等），这样会给文法引入一些多余的成分，对后续阶段造成不利影响，甚至会使各阶段变得混乱。因此，很多编译器（包括 GJC）经常要独立地构造语法分析树，为前、后端建立一个清晰的接口。

抽象语法树的构建可以采用已有的开源工具，也可以根据需求自行实现。笔者采用 Yacc 和 lex，以及 python 相关工具包，实现了一个 C 语言的抽象语法树解析工具。例如 C 语言源码片段如图 4 所示。

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void convert(int thousands, int hundreds, int tens, int ones)
{
char *num[] = {"", "One", "Two", "Three", "Four", "Five", "Six",
        "Seven", "Eight", "Nine"};

char *for_ten[] = {"", "", "Twenty", "Thirty", "Fourty", "Fifty", "Sixty",
      "Seventy", "Eighty", "Ninty"};

char *af_ten[] = {"Ten", "Eleven", "Twelve", "Thirteen", "Fourteen",
     "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Ninteen"};

  printf("\nThe year in words is:\n");

  printf("%s thousand", num[thousands]);
  if (hundreds != 0)
    printf(" %s hundred", num[hundreds]);

  if (tens != 1)
    printf(" %s %s", for_ten[tens], num[ones]);
  else
    printf(" %s", af_ten[ones]);
}
```

图 4 C 语言样例

生成的 AST 片段如图 5 所示。

```
FuncDef:
  Decl: convert, [], [], []
    FuncDecl:
      ParamList:
        Decl: thousands, [], [], []
          TypeDecl: thousands, []
            IdentifierType: ['int']
        Decl: hundreds, [], [], []
          TypeDecl: hundreds, []
            IdentifierType: ['int']
        Decl: tens, [], [], []
          TypeDecl: tens, []
            IdentifierType: ['int']
        Decl: ones, [], [], []
          TypeDecl: ones, []
            IdentifierType: ['int']
      TypeDecl: convert, []
        IdentifierType: ['void']
  Compound:
    Decl: num, [], [], []
      ArrayDecl:
        PtrDecl: []
          TypeDecl: num, []
            IdentifierType: ['char']
      InitList:
        Constant: string, ""
        Constant: string, "One"
        Constant: string, "Two"
        Constant: string, "Three"
        Constant: string, "Four"
        Constant: string, "Five"
        Constant: string, "Six"
        Constant: string, "Seven"
        Constant: string, "Eight"
        Constant: string, "Nire"
    Decl: for_ten, [], [], []
      ArrayDecl:
```

图 5 AST 解析样例

7 函数调用关系的展现

7.1 自行定义函数调用关系的展现

解析获得的调用关系，可以将此种关系使用数据库存储起来，然后根据需求进行展示。

7.2 使用业界标准与开源工具进行函数调用关系展现

另外可以将关系生成通用的展示格式，比如将其转换为图形文本文件。然后选择现有的工具将该图形文本文件转换为静态函数调用流程图。由于要体现函数体中被调用的函数间的前后顺序，因此要求用有向图来体现节点间的前后顺序。.dot 文件是描述有向图的文本文件。画图工具 graphviz 能将 .dot 文件转换为有向图。

结束语 前文给出了 3 种函数调用关系的提取方法，这里就这些方法进行简单的对比。从实现方法的简便性来说，使用现有的开源工具 ctags 最为简便，只需使用程序自动化解析并存储 ctags 生成的中间文件即可。但是此种方法的缺点也很明显，就是遇到源码量很大的项目时，不能非常准确地给出调用关系。所以另一方面，从准确性上来说，构建抽象语法树更能解决问题。

另外本文还给出了两种生成图关系的方法，同样地，在静态代码分析时可以根据需求进行使用。

参 考 文 献

[1] Wichmann B A, Canning A A, Clutterbuck D L, et al. Industrial Perspective on Static Analysis[J]. Software Engineering Journal 1995, 10(2): 69-75

[2] Briand P, Brochet M, Cambois T, et al. Software Quality Objectives for Source Code[C] // Proceedings Embedded Real Time Software and Systems 2010 Conference. ERTS2, Toulouse, Fra, 2010

[3] Improving Software Security with Precise Static and Runtime Analysis[D]. Stanford University, 2006