计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.190600042

基于关键词 Trie 树的 GCC 抽象语法树消除冗余算法

韩 磊 胡建鹏
上海工程技术大学电子电气工程学院 上海 201620
(m025117123@sues.edu.cn)

摘 要 GCC(GNU Compiler Collection)编译器编译 C 语言源程序所生成的抽象语法树文本中包含大量与源代码无关的冗余信息,若直接进行解析,会严重影响分析效率,降低分析精确度,同时会占用大量存储空间。针对此问题,提出一种基于关键词 Trie 树的 GCC 抽象语法树消除冗余算法,其根据包含抽象语法树文本有用信息节点的关键词建立 Trie 树,可实现对抽象语法树文本无用节点的过滤,从而达到优化编译的效果。相比传统 KMP 消除冗余算法,关键词 Trie 树算法可以有效避免去冗余过程中常量、变量等有用信息节点的丢失,确保数据的完整性;同时,关键词 Trie 树算法可以最大限度地减少重复前缀或后缀字符串的比较次数,节省了时空开销。挑选不同长度的 C 语言源码文件进行去冗余实验,测试该算法的性能,并将其与传统 KMP 算法进行对比。实验结果表明,所提算法的去冗效率和查准率均得到了极大的提高。

关键词:GCC;抽象语法树;关键词 Trie 树;优化编译;KMP;消除冗余
中图法分类号 TP311

Deduplication Algorithm of Abstract Syntax Tree in GCC Based on Trie Tree of Keywords

HAN Lei and HU Jian-peng
School of Electronic and Electrical Engineering,Shanghai University of Engineering Science,Shanghai 201620,China

Abstract The abstract syntax tree text generated by GCC compiler compiling C language source program contains a lot of redundant information independent of source code. If directly parsed,it will seriously affect the analysis efficiency,reduce the analysis accuracy,and occupy a large amount of storage space. Aiming at this problem,a GCC abstract syntax tree elimination redundancy algorithm based on the keyword Trie tree is proposed. The Trie tree is built according to the keywords containing the abstract syntax tree text useful information nodes,which can filter the useless node information of the syntax tree text,thus achieving optimized compilation results. Compared with the traditional KMP redundancy elimination algorithm,the keyword Trie tree algorithm can effectively avoid the loss of useful information nodes such as constants and variables in the process of redundancy removal and ensure the integrity of data. At the same time,the keyword Trie tree algorithm can minimize the comparison of repeated prefixes or suffix strings,saving time and space overhead. This paper selects different lengths of C language source files for deredundancy experiments,tests the performance of the algorithm,and compares it with the traditional KMP algorithm. The experimental results show that the algorithm can greatly improve the redundancy efficiency and precision.

Keywords GCC,Abstract syntax tree,Keyword Trie tree,Optimized compilation,KMP,Eliminate redundancy

1 引言

GCC 编译器是由 GNU 开发的编程语言编译器,能够支持 C,C++,Fortran 和 Java 等主流程序设计语言,被多种 Unix 操作系统如 Linux 和 Mac OS X 等采纳为标准编译器。

抽象语法树(Abstract Syntax Tree,AST)是编译系统中最常见的一种以树形结构为表现形式的中间表示,用来对前端语言的源代码进行规范化的抽象表示。GCC AST 作为一种良好的中间表示,包含显示源程序结构所需的全部静态信息,并且具有较高的存储效率[1]。抽象语法树的用途十分广泛,在编程语言的词法语法分析阶段[2]、静态代码抄袭检测[3-4]、程序切片算法提取特征[5]、检测缓冲区溢出漏洞[6]等程序分析领域占据着至关重要的地位。

GCC 抽象语法树文本文件是指 C 语言源文件经过 GCC 编译器编译后产生的以文本结构进行存储的语法树文件。GCC 在对源码的编译过程中,会对每一个源文件生成一个抽象语法树文本文件,使用编译指令“gcc-fdump-translation-unit-C”或者“gcc-fdump-tree-all”均可得到以“源码文件名

到稿日期:2019-06-11 返修日期:2019-11-20 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家自然科学基金项目(61802252);上海工程技术大学金课培育项目(DQY19004)
This work was supported by the National Natural Science Foundation of China(61802252) and Shanghai University of Engineering Science Golden Course Cultivation Project(DQY19004).
通信作者:胡建鹏(mr@sues.edu.cn)

001t.tu"为文件名的抽象语法树文件。图1给出了某源码抽象语法树文本的部分节点。

```
@1   type_decl   name:@2   type:@3   chain:@4
@2   identifier_node   strg:int   lngt:3
@3   integer_type   name:@1   size:@5   algn:32
       prec:32   sign:signed   min:@6
       max:@7
@4   type_decl   name:@8   type:@9   chain:@10
@5   integer_cst   type:@11   int:32
@6   integer_cst   type:@3   int:-2147483648
@7   integer_cst   type:@3   int:2147483647
@8   identifier_node   strg:char   lngt:4
@9   integer_type   name:@4   size:@12   algn:8
       prec:8   sign:signed   min:@13
       max:@14
@10   type_decl   name:@15   type:@16   chain:@17
```

图1 GCC抽象语法树文本

Fig. 1 GCC abstract syntax tree text

GCC抽象语法树以节点为单位来进行数据存储。表1列出了GCC抽象语法树文本文件的常见7个节点类型信息，每个节点由节点编号(@num)、节点标识符(节点类型)、子节点信息组成。每一个节点的节点编号是其唯一标识符，并且GCC抽象语法树文本文件是根据节点编号连续且升序排列存储的。

表1 抽象语法树常见的节点类型[1]

Table 1 Common node type of abstract syntax tree[1]

节点类型	后缀标识	引用示例
常量节点	_cst	Integre_cst
类型节点	_type	Array_type
声明节点	_decl	Var_decl
语句节点	_stmt	If_stmt
表达式节点	_expr	Decl_expr
标识符节点	_identifier	Ht_idenitify
列表节点	_list	Statement_list

2 研究背景与相关工作

2.1 研究背景

由于GCC编译源码生成抽象语法树文本文件是为了完成语法结构以及寄存器转移语言的转化过程，因此在设计之初就考虑到了底层优化等功能，也因此增加了大量中间节点。此举虽然一定程度地提高了完备性及优化性，但是随之而来的大量冗余信息却导致该文本不利于直接分析以及生成AST。在实际实验中，一个6行的打印"Hello World"的C语言源码会生成514kB的抽象语法树文件，这其中包含了大量编译中间节点信息用以辅助编译。例如，抽象语法树文件会默认引入预处理指令#include中的全部函数、结构体、变量等编译信息，其中多数信息在源码中没有涉及，也不利于代码的后续分析，其增加了对源代码分析的内存消耗，降低了分析效率。

2.2 相关工作

国内外对消除GCC抽象语法树文本文件冗余的研究较少，文献[7]最早提出了这一问题，但是没有提出解决方案。Li等[8]提出了一种基于宽度优先遍历的去除冗余算法，并取得了一定的效果，但是消除冗余的效率不高，需多次遍历文本节点，标记节点类型。算法耗时的原因在于使用KMP算法进行多次遍历，并且算法中将初次遍历后标记为useful_node节点的子节点仅标记为unknown_node节点，这就丢失了许多常量、变量类型的节点信息。

Tian等[9]在此算法基础上做出了改进，虽增加了子节点遍历时间，但相比其他算法消除GCCAST冗余还是节省了时空开销，消除返祖边去冗余简化GCC抽象语法树算法同样多次遍历节点类型以查找有用信息节点，同时还须返回遍历查找误消除的节点——call_expr(函数调用语句)节点，这就导致了算法效率低下，且算法繁复。

2.3 Trie树算法简介

Trie树(又名字典树、关键词查找树)[10-12]是实现网络信息检索的一种有效算法，经常被各大搜索引擎用来查找关键词和统计文本词频等。由于许多字符串数据都拥有相同的前缀或者后缀字符串，同时Trie树能够最大限度地减少重复字符串前缀或后缀的比较次数，从而节省了时空开销。考虑到时空效率、冗余消除，并为了确保数据准确性等，本文提出基于有用节点关键词的Trie树算法[13-16]来消除GCC抽象语法树的冗余信息。

3 算法设计

算法的整体流程为：1)通过GCC编译器编译不同大小的C语言源码文件，得到原始抽象语法树数据；2)通过文本预处理将一个节点的信息(包括其描述信息和子节点信息)作为一个单元存储；3)构造GCC抽象语法树文本文件经编译产生的有效节点类型，以及与源文件相关的有用节点关键词词库；4)创建有用节点信息关键词Trie树；5)对进行文本预处理的GCC抽象语法树文本进行有效信息关键词的检索，并将检索到的包含有效信息字段的节点输出；6)将上一步输出的节点的子节点信息也输出，得到消除冗余后的GCC抽象语法树文本。基于关键词Trie树消除GCC抽象语法树文本冗余的整体流程如图2所示。

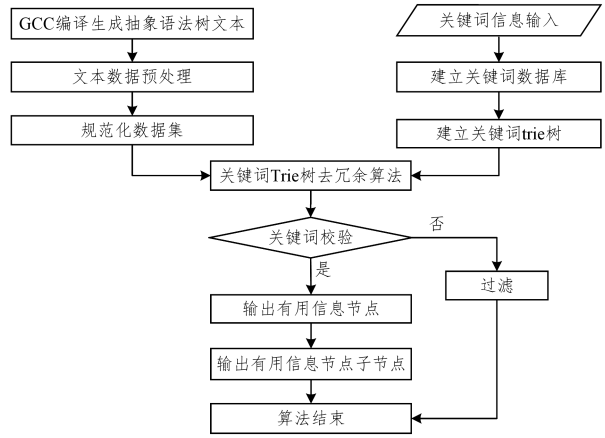


图2 消除冗余的整体流程图

Fig. 2 Overall flow chart for eliminating redundancy

3.1 文本预处理

生成GCC抽象语法树文本文件的实验环境如下：Linux版本为centos6.8，GCC版本为6.5.0。根据GCC抽象语法

树节点自上而下递增存储的逻辑结构，可以容易地根据其唯一标识符对其进行处理，将一个节点的数据存储为一个单元。对于初步处理的数据，依据有用信息节点的类型及关键词构建关键词词库。

3.2　有用节点关键词 Trie 树的构建算法

定义 1（关键词）　GCC 抽象语法树文本中描述节点的信息中包含有用节点信息的关键词。

GCC 抽象语法树的节点是否有用，主要是通过判定该节点是否包含固定类型的节点信息，如“call_expr”“real_cst”等，以及节点描述信息是否包含源文件名称字符，如“srcp：源文件名”。将这些关键词进行建库，包含这些字段信息的节点描述信息即被认定为是有用信息节点，当遇到包含“srcp”字段但是不包含源文件名的字段时，算法能将其检测出来并判定为冗余信息。

Trie 树算法的构建（见算法 1）需将前文构造的关键词词库作为输入项，从关键词的数据库中读取关键词，然后将读取的每个关键词添加到关键词 Trie 树中。算法 1 中，为了提高关键词信息检索时的查找效率，构建决策树时将有用节点关键词的第一个关键字母按照英文字母表进行排列。构建关键词 Trie 树的整体结构如图 3 所示。这样，在检索到相同前缀的关键词时，仅需将其存储到相同的前驱节点，这不仅能有效减少存储节点，利于降低空间复杂度，而且在对 GCC 抽象语法树文本进行冗余优化时可以节省时间，提高优化效率。

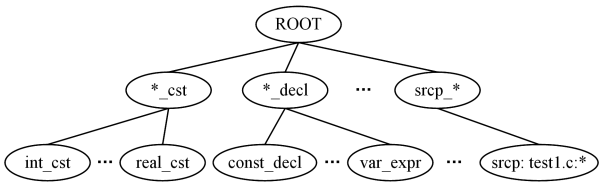


图 3　关键词 Trie 树的整体结构

Fig. 3　Overall structure diagram for keywords Trie tree

以常见声明节点为例，图 4 给出了以 decl 为后缀关键词构造的声明节点 Trie 树示例。其中，“*”表示该关键词的子节点字符串，主要包含常量声明（CONST_DECL）、成员声明（FIELD_DECL）、函数声明（FUNCTION_DECL）、标号声明（LABEL_DECL）、变量声明（VAR_DECL）等。

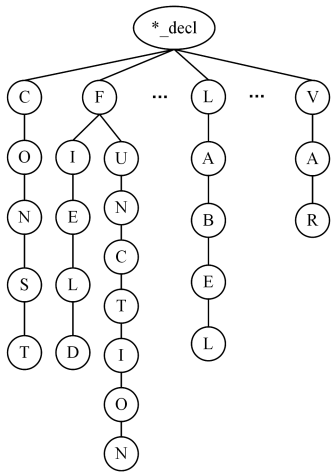


图 4　声明节点 Trie 树示例

Fig. 4　Example of declaring node Trie tree

定义 2（关键词集合）　关键词集合表示为：$K=\{k_1,k_2,\cdots,k_i,\cdots k_n\}(1\leqslant i\leqslant n)$。其中，$n$ 表示类型节点关键词的个数；k_i 表示第 i 个类型节点关键词，$k_i=\{k_{i1},k_{i2},\cdots,k_{ij},\cdots,k_{i,m}\}(1\leqslant j\leqslant m)$，$k_{ij}$ 表示第 i 个类型节点关键词的第 j 个子节点关键词，m 表示子节点的个数。

定义 3　关键词 Trie 树中的某个节点为 C，其包含的有用信息子节点集合为 C_N，其中第 q 个子节点记为 C_{Nq}。

算法 1　关键词 Trie 树构建算法

输入：GCC 抽象语法树文本中包含有用节点信息的关键词字段集合 K

输出：GCCAST 关键词 Trie 树

1. 初始化检测信息：i＝1，j＝1，q＝1，q 记录子节点的序号，C_i＝ROOT，C_i 为正在检测的节点。
2. 若 i≤n，获取类型节点关键词的首字母 S。
3. 进入到 S 子节点树查询，将 K_{ij} 与类型节点关键词首字母 S 的第 q 个子节点 $child_q$ 进行比较。
4. 若 $K_{ij}==child_q$，则 j＋＋。

   此时：若 j＜m，S＝$child_q$，q＝1，返回步骤 3；若 j≥m，i＋＋，当 i≥n 时，结束算法；

   当 i＜n 时，返回步骤 2。
5. 若 $K_{ij}\neq child_{ij}$，查询 $child_q$ 的下一节点是否为空值。
6. 若 $child_q$ 下一节点为空，则创建新节点 $child_{q+1}$，并将 K_{ij} 的值赋给 $child_{q+1}$，j＋＋。

   此时：若 j＜m，则创建子节点，并赋值 $child_{ij}$，j＋＋；若 j≥m，则初始化，i＋＋，当 i＜n 时，返回步骤 2；当 i≥n 时，算法结束。
7. 若 $child_q$ 的下一兄弟节点不为空，则 q＋＋，返回步骤 2，处理下一个关键词。
8. 算法结束。

在 GCC 抽象语法树文本中，所有节点信息都是以英文字符和阿拉伯数字组成，并不涉及中文字符；同样，根据算法的设计可以发现，关键词 Trie 树的高度与抽象语法树关键词库中关键词的数量有关，而关键词库的关键词数量有限，因此 Trie 树的高度对算法空间复杂度的影响微乎其微。而 GCC 抽象语法树文本的有用节点信息拥有相同的节点类型后缀词，并且 Trie 树只有两层节点，这大大降低了空间复杂度，简化了查询过程，提高了算法效率。

3.3　关键词 Trie 树算法消除冗余

将经过处理的 GCC 抽象语法树文本作为输入，用关键词 Trie 树算法（见算法 2）对其进行优化，去除冗余信息。算法 2 中，在检测关键词的过程中，如果一个节点的描述信息中包含多个关键词，例如同时包含源文件（源文件名 text1. c）节点信息的关键字“srcp：test1. c. ＊ ”和函数调用语句节点信息关键字“call_expr”时，则按照最大匹配的原则进行查找。消除冗余后的 GCC 抽象语法树文本更加纯净，数据更加完整。

定义 4　GCC 抽象语法树文本数据流表示为：$AST_{test}=\{ast_1,ast_2,ast_3,\cdots,ast_i,\cdots,ast_n\}(1\leqslant i\leqslant n)$。其中，$n$ 为抽象语法树文本节点数，ast_i 表示文本中的节点描述字符信息。

算法 2　基于关键词 Trie 树算法消除冗余

输入：GCC 抽象语法树文本数据流 AST_{test}，GCC AST 关键词 Trie 树

输出：优化冗余后的 GCC 抽象语法树文本 AST_{new}

1. 初始化 i＝1，q＝1，q 用于记录进入分支的字符序列号。

2. 输入ast_i，若 i≥n，转步骤 7，否则将 i 的值赋给 q，j=1，获取ast_i的节点类型首字母 S。
3. 查询关键词 Trie 树中节点类型首字母为 S 的子节点$child_j$，并将其与ast_i进行匹配。
4. 若ast_i==$child_j$，输出该条ast_i所在唯一标识行号@number 及其节点信息至AST_{new}。
5. 将输出的ast_i描述字段中的子节点信息输出至AST_{new}。
6. 若ast_i≠$child_j$，查询下一节点是否为 Null，若下一节点不为 Null，则转至步骤 3。
7. 若下一节点为 Null，则 i=q+1。
   此时：若 i<n，转至步骤 2；
   若 i≥n，算法结束。
8. 算法结束。

4 实验分析与算法评估

4.1 实验准备

实验环境的配置如表 2 所列。

表 2 实验环境说明

Table 2 Experimental of environment description

实验环境	软硬件	型号版本
硬件环境	CPU	I7-2600，3.40 GHz
	内存	8 GB
	硬盘	140 GB
	系统	Win7 64 位
软件环境	开发语言	Python
	开发软件	PyCharm
	版本	3.6.4

为了测试算法消除冗余的性能，选取 5 段长度不等的 C 语言源码进行优化，分别记录 C 源码在 GCC 编译后产生的节点数以及消除冗余后的节点数。在 centos6.8 下，经过版本为 6.5.0 的 GCC 编译器编译后生成抽象语法树文本文件，其中：

程序 1 《计算自由落体反弹高度》

程序 2 《求一元二次方程的根》

程序 3 《回归分析求值》

程序 4 《冒泡排序》

程序 5 《函数迭代问题》

程序 1－程序 3 来自文献[17]；程序 4 和程序 5 来自文献[18]。

4.2 实验结果分析

对基于有用信息关键词 Trie 树去除 GCC 抽象语法树文本冗余算法进行性能评估。衡量算法的指标之一为优化率，其计算方法为优化率＝(无用信息节点个数/抽象语法树文本节点总数)×100%。实验结果如表 3 所列，可以看出所选取的 5 段长度不等的源码文件的优化率均在 90%以上。

表 3 Trie 树消除冗余性能

Table 3 Performance of Trie tree to eliminate redundancy

程序编号	源码行数	消除前节点数	消除后节点数	优化时间/s	优化率/%
1	15	4439	116	0.073	97.4
2	22	5017	243	0.083	95.2
3	55	5159	335	0.087	93.5
4	65	4625	284	0.074	93.9
5	85	4690	356	0.075	92.4

文献[8]提出的简化 GCC 抽象语法树文本算法的核心是 KMP 算法(在图 5、图 6 中命名为 KMP1)，而文献[9]提出的先删除返祖边冗余后简化 GCC 抽象语法树文本的算法同样采用了 KMP 算法(在图 5、图 6 中名为 KMP2)。

将本文算法与文献[8]和文献[9]中的算法进行对比，选取相同的 5 组 C 源程序经由 GCC 编译生成的抽象语法树文本文件作为实验对象，分别采用 3 种算法对其进行冗余消除。如图 5 所示，文献[8]中的算法存在第一次遍历后将 useful_node 的所有子节点均标记为 unknown_node 节点的错误，因此其损失的节点数最多；文献[9]中的算法在前期遍历标记时存在误消节点错误(函数调用语句节点 call_expr)，而在后期找回误消节点时同样会损失这些误消节点的子节点；而基于有用信息关键词 trie 树算法则有效避免了简化过程中有用信息(包括常量和变量)丢失和误消的情况，确保了数据的完整性。从图 5 可以看出，随着测试文本数据节点数的增加，文献[15]和文献[16]中的算法所丢失的节点数据越来越多。

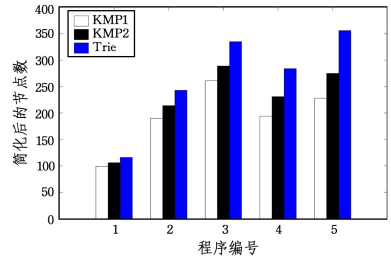


图 5 3 种算法消除冗余后的节点数

Fig. 5 Number of nodes after eliminating redundancy by three algorithms

3 种算法的耗时对比结果如图 6 所示。文献[8]提出的算法由于须对节点类型进行判断，对文本进行了多次遍历，因此耗时最长；而文献[9]中的算法在去除返祖边后同样采用 KMP 算法进行了多次遍历，但时耗低于文献[8]中的算法；而基于关键词 Trie 树算法仅须遍历一次，因此时耗最短。

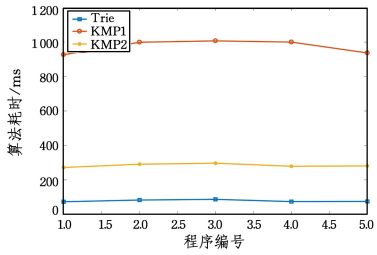


图 6 算法时耗对比

Fig. 6 Time-consuming comparison of algorithms

综合图 5 和图 6 可知，在相同实验环境下，关键词 Trie 树算法无论是查全率还是时耗，均优于文献[8]和文献[9]的 KMP 算法，能在保证查准率的同时减少时空开销。

4.3 算法复杂度评估

关键词 Trie 树算法直接使用包含 GCC 抽象语法树有用信息的关键词生成的 Trie 树，对抽象语法树文本信息进行冗余消除，采用 Trie 树的方法代替了传统 KMP 算法消除冗余，实现了数据的分流；依据抽象语法树文本不包含中文字符，以及文本中的节点编号是连续且升序排列存储的特性，设计出

更加适合的关键词 Trie 树算法。关键词 Trie 树算法的使用，降低了匹配查询的复杂度，提高了检索效率。

关键词 Trie 树消除冗余算法的时间复杂度与 Trie 树的树高 h 以及抽象语法树文本词汇总量 m 相关，为 $O(h\times m)$。空间复杂度取决于决策树的大小，为 $O(n)$，其中 n 为关键词决策树词库中的总条数。算法使用了 Trie 树与 GCC 抽象语法树文本数据流最大匹配的方式，因此可以有效找出有用节点信息，优化率较高，同时该算法不会受到抽象语法树文本和 Trie 树大小变化的影响。

4.4　算法局限性

本文算法也存在有一定的局限性。关键词 Trie 树的词库需要人为构建添加，关键词无法自动更新，需要人为定期地将关键词添加到关键词词库；同时，随着源代码数量的增加，关键词 Trie 树算法消除冗余的时耗大幅增加，程序运行速度降低，因此优化算法结构、改进算法语言至关重要。

结束语　经过对 GCC 抽象语法树文本特点及传统消除冗余算法的研究，本文提出了一种基于 GCC 抽象语法树的有用信息关键词 Trie 树算法。该算法将有用信息关键词构建成分流 Trie 树，利用构建的关键词 Trie 树对抽象语法树文本进行简化，以消除文本中包含的大量冗余信息。对比实验表明，关键词 Trie 树算法的简化率达到了 90%以上，并且算法时耗也比传统 KMP 算法低得多，能以较小的时空代价消除 GCC 抽象语法树文本冗余。下一步的工作包括：改进算法实现以动态更新关键词，优化关键词 Trie 树算法的结构，提高算法的性能。

参 考 文 献

[1] 王亚刚. 深入分析 GCC[M]. 北京：机械工业出版社，2017.

[2] GCC. The GNU compiler collection [EB/OL]. [2018-10-26]. http://gcc.gnu.org.

[3] ZHANG T，CHEN J，JIANG H，et al. Bug Report Enrichment with Application of Automated Fixer Recommendation[C]// 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE Computer Society，2017.

[4] VISHWACHI，GUPTA S. Detection of near-miss clones using metrics and Abstract Syntax Trees[C]// International Conferenceon Inventive Communication and Computational Technologles (ICICCT). New York：IEEE，2017：230-234.

[5] YANG X C，JIANG J，MA X D，et al. Program slicing technology based on critical variable dataflow analysis algorithm in GCC compiler[J]. Computer Engineering and Applications，2017，53(24)：40-47，54.

[6] YANG X L，LIU J. Statically Detecting Likely Buffer Overflow Vulnerabilities in C/C++ Program[J]. Computer Engineering and Applications，2004(20)：108-110.

[7] ANTONIOL G，PENTA M D，MASONE G，et al. XOgastan：XML-oriented gcc AST analysis and transformations[C]// Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. IEEE，2003.

[8] LI X，WANG T T，SU X H，et al. Research on Eliminating Redundancies of GCC AST Test[J]. Computer Science，2008(10)：170-172.

[9] TIAN B C，SUN K，CHAO H Q. New Algorithm of Simplifying GCC Syntax Tree[J]. Computer Science，2015，42(S1)：516-518，530.

[10] ZHANG Q，JIN Y C，LI M，et al. Implementation of Trie Tree Router Lookup Algorithm in Network Processor[J]. Computer Engineering，2014，40(4)：98-102.

[11] XU G T，ZHANG M. Research on Trie Tree Keeyword Fast Matching Algorithm in Network Security Situational Awareness [J]. Netinfo Security，2019(4)：55-62.

[12] LI A，HE D，WANG H. An advanced trie-based HTTP parsing algorithm[C]// Sixth International Conference on Information Science & Technology. IEEE，2016.

[13] YANG T，MI Z，DUAN R，et al. An ultra-fast universal incremental update algorithm for trie-based routing lookup[C]// 2012 20th IEEE International Conference on Network Protocols (ICNP). IEEE Computer Society，2012.

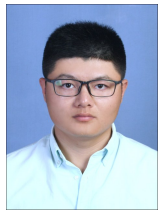
[14] XUE P Q，XIAN Y，NURBOL ，et al. Sensitive information filtering algorithm based on Uyghur text information network research[J]. Computer Engineering and Applications，2018，54(5)：236-241，246.

[15] GONG P，OSBORN W. A Compact-Trie-Based Structure for K-Nearest-Neighbour Searching[C]// IEEE International Conference on Advanced Information Networking & Applications. IEEE，2017：578-585.

[16] GHASEMI C，YOUSEFI H，SHIN K G，et al. A Fast and Memory-Efficient Trie Structure for Name-Based Packet Forwarding [C]// 2018 IEEE 26th International Conference on Network Protocols (ICNP). IEEE，2018.

[17] HUANG R，ZHAO Y. C Programming[M]. BeiJing：Tsinghua University Press，2012.

[18] DEITEL P，DEITEL H C. How To Program International Edition，Seventh Edition[M]. Beijing：Publishing House of Electronics Industry，2018.



HAN Lei，born in 1993，postgraduate，is a member of China Computer Federation. His main research interests include machine learning，data mining.

HU Jian-peng，born in 1980，associate professor，is a member of China Computer Federation. His main research interests include software engneering，service computing and data mining.