

一种基于指针逻辑的代码安全属性分析方法

张 阳¹⁾ 程 亮²⁾

¹⁾(中国科学院软件研究所 北京 100190)

²⁾(中国科学技术大学电子工程与信息科学系 合肥 230027)

摘 要 在分析和总结前人工作的基础上,提出了一种改进的代码安全属性验证方法.该方法在利用传统的源代码安全属性验证工具的基础上,加入了指针逻辑,针对现有代码属性分析技术只能对C语言子集进行分析验证的不足,利用指针逻辑对源代码的分析结果对源代码中的指针进行替换,从而避开了传统静态代码属性验证工具对指针处理功能太弱的瓶颈,可以实现对C语言中的部分指针及运算进行处理.

关键词 操作系统安全;形式化验证;代码分析;模型检测;指针逻辑

中图法分类号 TP309

DOI号: 10.3724/SP.J.1016.2009.01119

A New Property Verification Method for Code Security Based on Pointer Logic

ZHANG Yang¹⁾ CHENG Liang²⁾

¹⁾(Institute of Software, Chinese Academy of Sciences, Beijing 100190)

²⁾(Department of Electronic Engineering and Information Science, University of Science and Technology of China, Hefei 230027)

Abstract This paper proposes an improved verification method for code security property on the basis of the study of predecessors' work. According to the situation that current code property verification can only deal with a subset of C programming language, this paper introduces the pointer logic, whose result can be used by the replacement algorithm to substitute all the pointers in the source code, to the conventional code security property verification tools. The proposed approach avoids the weakness of pointer processing in the traditional static code property verification, and therefore can handle pointers in C programming language when property verification partially.

Keywords operating system security; formal verification; code analysis; model checking; pointer logic

1 引 言

近几年来发生了多起大规模的安全事件,大部分是由于操作系统安全漏洞所造成的全球大量计算机被感染,造成了重大经济损失.由此,使用自动化的方法从操作系统的源代码出发分析漏洞的存在与否成为当今国际上的一个研究热点.代码(程序)分析的目的在于检验系统的实际代码是否确实满足或

违背了规范中所声明的安全准则.传统的代码分析方式以代码走查(walkthrough)为主,纯手工劳动,费时费力,而且效率低下.目前国际上许多大学和公司都开展自动化的软件错误检查方法的研究,其理论基础包括了模型检测技术(model checking)^[1]、定理证明技术(theory proving)^[2]以及约束求解技术(constraint solving)^[3].

处于效率的考虑,操作系统代码多半用C语言写成.目前,针对C语言的静态代码属性验证工具

多半采用对源代码抽象进行模型检测以发现其中违反待验证安全属性的反例(路径)的办法来实现属性的验证. 由于模型检测器生成的反例是基于源代码的抽象模型的, 因此, 程序抽象技术的好坏就直接影响到反例的正确性. 然而, 由于 C 语言中指针的应用非常灵活, 多级指针、栈变量指针、运行时内存分配、强制类型转换、函数指针、外部变量指针、地址操作符、结构体和联合体等种种指针的应用使得在程序抽象时处理起来非常困难, 因此, 大部分的静态代码属性验证工具, 如 BOOP^[4]、BLAST^[5] 等均没有能对指针作处理, SATABS^[6] 则仅能发现对 Null 指针的脱引用.

而另一方面, 基于程序分析理论和方法的代码查错工具, 如 ESC(Extended Static Checking)^[7] 和 Spec#^[8] 等, 试图通过应用类型系统或者逻辑系统部分实现了对指针的分析处理. 然而这类代码差错工具或者只能应对 C 语言的子集, 对待验证程序提出各种限制^[9-10]; 或者应用太过单一, 仅仅能处理如空指针调用之类的内存安全错误^[11], 而且缺乏可靠性证明, 更多地被当作程序差错工具而不是属性验证工具. 比如, 作为专门的基于类型的内存安全分析工具的 CCured^[12], 虽然可以 C 语言中对部分指针实现静态分析, 但是它需要插入动态检查代码, 这在很多情况下会降低系统的效率.

Cyclone^[13] 为 C 设计新的类型系统, 重点是加入更多高级语言特性和类型. 它使用了基于区域的存储管理(region-based memory managment), 这种管理方式除了自身有较大局限性外, 还要求程序员在代码中加入对这些机制的支持代码, 这对历史上遗留的 C 代码来说工程浩大.

陈意云等人^[14]提出了一种 C 语言安全程序的设计和证明框架, 并设计了指针逻辑系统, 从而可以对程序进行精确的指针分析, 并提出了一种可证明安全性的类 C 语言 PointerC^[11]. 与之相类似的, Xi 等人的 ATS(Applied Type System) 项目^[15], 也将 Hoare 逻辑形式的断言引入类型系统, 在类型系统上模拟 Hoare 逻辑的部分推理. 但相对而言, 陈的方法使用单独的指针逻辑系统对副条件进行推理, 可以证明更强的程序安全属性, 而 ATS 的类型系统和 CCured 一样需要程序员显式标注. 此外 ATS 的类型检查的表达能力有限, 仅能推理整数等较简单论域的性质, 证明能力总体来说弱于指针逻辑.

通过对目前流行的指针分析工具(代码查错工具)的分析类比, 在吸取前人现有工作成果的基础

上, 提出一种改进的代码安全属性验证方法. 该方法先利用指针分析技术对代码进行分析, 获得代码中所有指针的指向信息, 然后根据获得的信息将源代码中的指针相关代码代换为非指针类型的代码, 再交由代码性质验证工具进行属性验证. 该方法的主要贡献在于将指针分析技术融入现有的静态代码属性验证工具中, 使得验证人员可以对更实际、广泛的 C 语言源程序进行安全属性的形式化验证, 扩展了现有工具的适用范围.

本文第 2 节是对方法框架的总体介绍; 第 3 和第 4 节分别详细阐述指针等价类的生成方法和指针替换算法; 第 5 节用一个实例说明该方法的有效性; 最后是结论以及下一步的工作计划.

2 方法框架描述

对于目前的属性验证工具来说, 重点是程序的属性分析, 如某个程序点是否可达, 多线程编程中是否存在互斥或者竞争状态等程序逻辑属性, 当涉及到指针及别名分析时, 不少工具采用了忽略^[4]或者简化^[5]的办法, 最终的代价是结果可能出现误报或者是实际应用工具前需要手工修改相应的代码. 指针分析工具则主要用来分析程序中空指针的出现或者内存泄漏等物理属性情况, 和其它静态技术类似, 指针分析受到可判定性问题的困扰, 对大多数语言来说, 所得到的解总是一个近似, 另外指针分析的方法和属性验证方法迥异, 从而使得指针分析的用途总受到一定的限制. 我们的方法综合了指针分析的最新成果和属性验证的经验, 试图将二者融合, 从而扩展属性验证的适用范围. 该方法的框架如图 1 所示, 主要包括了如下的 3 个部分:

(1) 利用指针逻辑来对 C 代码进行分析, 以得到指针别名的等价类集合. 在指向对象集合的精度上, 不同的应用要求不同的粒度. 不同的精度要求采用不同的分析方法, 有流敏感和流不敏感的区别, 路径敏感和路径不敏感的区别以及过程内和过程间的区别. 出于软件安全方面的要求, 我们要求源代码中的指针分析是精确的而不是近似的. 指针逻辑在不影响语言功能的情况下, 对 C 语言中难以判定的指针使用方式进行了限制, 解决了 Hoare 逻辑处理别名困难的问题, 而其可靠性也在文献^[11]中经过了严格的证明, 很好地实现了对指针的精确分析, 因此成为我们代码属性验证方法的首选.

(2) 根据指针分析得到的指针等价类集合, 对

源代码中的指针进行替换. 这一步既要求替换后的代码完全消除指针的引用, 还要保证属性验证可以获得完整的指针指向信息以及对象的类型信息. 由于指针逻辑的目的是为了证明程序是否满足内存安全策略等物理特性, 因此在利用指针逻辑遍历 C 语言源代码之后, 得到的等价类集仅仅包含程序运行到结束点时指针指向情况. 而属性验证则希望能获知指针在程序任一位置(程序点)的指向对象, 以便于在模型检测时能准确判断系统状态. 由此, 我们还需要对

指针逻辑做一定的修改以适应后端属性验证的要求.

(3) 将替换后的代码交由属性验证工具验证. 属性验证的基本原理是利用抽象工具得到 C 源代码的抽象模型, 这种抽象和编译器前端的工作类似, 事实上, 有些属性验证工具就是利用 gcc 前端来获得程序抽象的^[4]. 所以, 第二步中源代码中的指针均以其他类型替换之后, 程序的行为不但不会发生改变, 属性验证工具仍可以发挥其正常的功能, 而且扩大了属性验证所能探测的状态空间.

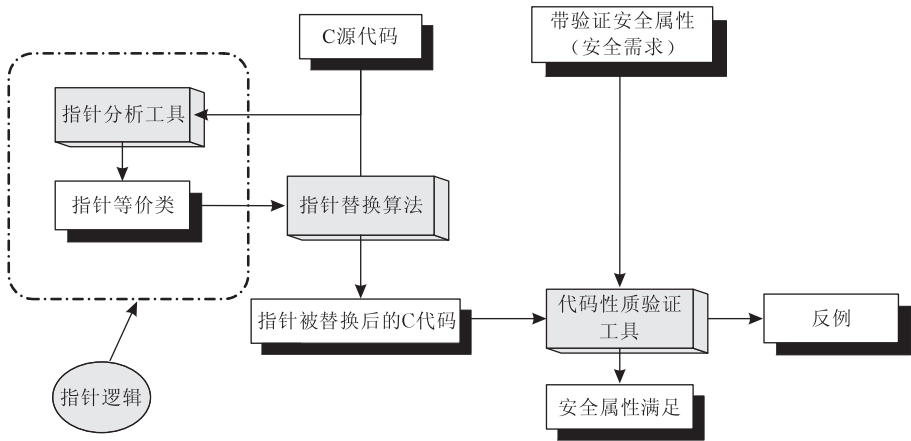


图 1 基于指针逻辑的代码安全属性验证框架图

指针逻辑的演算可以在编辑阶段由 PointerC 编译器和自动定理证明来完成^[11], 指针替换亦可以借助工具辅助完成, 因此我们的方法人工干预较少, 效率也比较高. 下面我们就分别介绍如何利用指针逻辑生成能够保存中间信息的指针等价类以及如何利用等价类来获得不含指针的等效源代码.

3 指针等价类生成

如上所述, 指针逻辑的主要功能是为了产生指针等价类, 由于我们的指针替换需要各个程序点的指针信息, 因此对于指针逻辑的推理规则需要加以修改. 下面我们详细介绍如何基于指针逻辑生成源代码的指针等价类以及对指针逻辑推理规则的修改.

3.1 指针逻辑简介

指针逻辑的引入, 原本是为了证明类 C 语言 PointerC 在指针使用上的安全性. 考虑到 C 语言中的指针太过灵活, PointerC 对指针运算做了一定的规定, 以保证程序运行时不会出现对 Null 指针或悬空指针(dangling pointer)进行存取指向对象的操作或者把 Null 指针或悬空指针作为 free 过程调用的实在参数、发生内存泄漏的行为, 从而破坏系统的可

靠性. 主要规定如下^[14]:

- (1) 允许指针类型的变量用于“=”赋值以及“==”、“!=”等布尔运算;
- (2) 允许指针的存取指向对象等运算;
- (3) 允许指针作为 free() 的参数;
- (4) 禁止指针算术和取地址运算(&);
- (5) malloc() 和 free() 为与定义函数, 不会失败, 且未释放空间不存在重叠.

上述限制的目的是为了便于静态检查程序的安全性, 因此我们的方法沿袭了这些限制. 程序运行时出现对 NULL 指针或悬空指针进行存取指向对象的操作、把 NULL 指针或悬空指针作为 free 函数调用的实在参数、发生内存泄漏等都被认为不足基本安全策略(类型安全和内存安全等). 该语言定型规则中的附加条件就是用来禁止这些情况的出现, 事实上, 指针逻辑的用途之一就是用来完成对这些附加条件的静态检查.

指针分析主要回答: 对于指针类型的变量, 它们在运行时可能指向的对象集合是什么. 在指针逻辑中, 程序点的 Null 指针集合用 N 表示, 悬空指针的集合用 D 表示, 有效指针集合用 Π 表示. Π 中指针的具体值并不重要, 重要的是它们是否相等, 因此基

于相等与否把它们划分成若干等价集合. 例如, 若 Π 中有等价集合 $\{p, q\}$, 则它表示 p 和 q 是相等的有效指针, 并且它们不等于其它集合中的指针. 一个等价集合不能删掉任何元素, 也不能分成若干子集, 因为这样做都会使指针信息发生变化. 因此, 在指针逻辑的断言演算中, Π 、 N 和 D 中的等价集合被看成命题常元, 由于它们在语法结构的前后条件中本质上是逻辑表达式, 因此用“ \wedge ”连接, 作为缩写, 有时表示为 $\Pi \wedge N \wedge D$.

3.2 对指针逻辑推理规则的修改

利用以上的指针逻辑, 我们可以对 C 语言源代码进行分析, 得到其中所有指针的等价类集合. 如前所述, 指针的等价类集合 Π 、 N 和 D 分别表示有效指针集合、空指针集合和悬空指针集合, 而实际上, 这些集合是指针等价类的并集. 如 $\Pi = \{s, t\} \wedge \{p, q\}$ 实际代表的是如下逻辑命题:

$$(s=t) \wedge (s \in effective) \wedge (p=q) \wedge (p \in effective) \wedge (p \neq s),$$

其中, $effective$ 为有效指针的集合.

由于属性验证在源代码抽象时会将指针部分剔除^[4]或者作简化处理^[5], 以保证后端模型检测能够完整处理代码抽象, 因此, 我们在做指针替换时需要将指针替换成它逆向引用 (dereference) 的数据类型, 与此相对应地, 该指针在程序中的声明也需要做同样的替换. 另外, 虽然指针逻辑可以分别得到有效指针、空指针和悬空指针的集合 Π 、 N 和 D , 但其目的是为了便于开发出具证明的原形语言的编译器 (certifying compiler)^[14]. 该编译器可以在利用这些信息将源代码翻译成目标代码的同时对程序中的指针是否存在违反安全策略进行证明. 因此, 我们在替换时需要将 Π 、 N 和 D 三者内全部的等价类都予以替换.

基于以上考虑, 在实际分析时, 我们可以采用如下的方法来记录指针.

定义 1. 对于 Ψ 中的指针 p , 都表示为一个三元组

$$p = \langle line, type, ap \rangle, \forall p <: \Psi,$$

其中,

(1) $line \in Lines$, 表示 p 当前在程序中的行号, $Lines$ 为行号集合;

(2) $type \in Types$, 表示 p 指向的数据类型, $Types$ 为 C 语言中的基本数据类型;

(3) $ap \in AccessPaths$, 表示 p 所对应的访问路径, $AccessPaths$ 为访问路径集合.

对于所有的 $p <: \Pi$, 都有 $p \in effective$, 其中, $effective$ 为有效指针集合. 因为源程序中的指针赋值可能会导致同一个指针前后指向不同的对象, 此处引入行号集合 $Lines$ 记录指针出现的位置, 以保证指针替换时不会将不同的对象混淆.

我们的目的是为了通过指针逻辑获得源代码中的指针等价类, 每一个等价类对应一个内存对象. 由于在执行文献[14]的指针分析后得到的等价类只能反映程序结束点处的状态, 对于后续的代码属性验证来说, 这样丢失了中间信息的等价类集合是无法准确反映出程序运行时的数据流向的, 是不可用的. 因此, 需要对指针逻辑的公理和推理规则进行修改, 以保存中间信息.

指针逻辑中共有 6 类推理规则, 其中影响 Ψ 的规则有

(1) 指针之间的赋值 $p = q$. 该规则又针对 p 、 q 属于不同集合细分为 6 种情况, 为了同时保留赋值前后的等价类信息, 我们修改推理规则如下:

① p 和 q 是相等的有效指针, 或都等于 NULL, 此时赋值动作对等价类集合并无影响, 即

$$\frac{\exists S: \Pi. (p <: S \wedge q <: S) \vee (p <: N \wedge (q <: N \vee q \equiv \text{NULL}))}{\{\Pi \wedge N \wedge D\} p = q \{\Pi \wedge N \wedge D\}}.$$

② p 和 q 是不相等的有效指针, 此时赋值会将 p 从原来的等价类中转移到 q 所在的等价类, 由于在后端做属性分析时需要保留 p 在所有程序点所指向的对象, 因此, 不能将 p 从 Π 中删除, 即

$$\frac{\exists S_1: \Pi. \exists S_2: \Pi. (S_1 \neq S_2 \wedge p <: S_1 \wedge q <: S_2) \wedge \neg leak(p)}{\{\Pi \wedge N \wedge D\} p = q \{(\Pi + (equals(p) \cup q) \wedge N \setminus p \wedge D) \setminus p\}}.$$

需要注意的是, 结论后件中的 Π 和 $equals(p) \cup q$ 虽然都包含指针 p , 但由于二者的行号不同, 仍作为两个不同的变量.

③ p 是 NULL 指针, q 是有效指针时, 同②有

$$\frac{q <: N \wedge q <: N}{\{\Pi \wedge N \wedge D\} p = q \{(\Pi + (equals(p) \cup q) \wedge N \setminus p \wedge D)\}}.$$

④ p 是悬空指针, q 是有效指针时:

$$\frac{q <: D \wedge q <: N}{\{\Pi \wedge N \wedge D\} p = q \{(\Pi + (equals(p) \cup q) \wedge N \setminus p \wedge D) \setminus p\}}.$$

⑤ p 是有效指针, q 等于 NULL 时:

$$\frac{p <: \Pi \wedge (q <: N \vee q \equiv \text{NULL}) \wedge \neg leak(p)}{\{\Pi \wedge N \wedge D\} p = q \{\Pi \wedge N \setminus p \cup \{p\} \wedge D \setminus p\}}.$$

⑥ p 是悬空指针, q 等于 NULL 时:

$$\frac{p <: D \wedge (q <: N \vee q \equiv \text{NULL})}{\{\Pi \wedge N \wedge D\} p = q \{\Pi \wedge (N \cup \{p\}) \wedge D\}}.$$

(2) 分配空间语句 $p = \text{malloc}(T)$, 其中 T 是类

型. 若 T 是结构类型, r_1, \dots, r_n 是其中的指针域在该类型中的访问路径.

$$\frac{p < : N}{\{\Pi \wedge N \wedge D\} p = \text{malloc}(T) \{(\Pi + p) \wedge N \wedge (D \cup \{p \rightarrow r_1, \dots, p \rightarrow r_n\})\}}$$

(3) 释放空间语句 $\text{free}(p)$, 此处涉及到 p 所指向对象的销毁, 也不能从 Π 中删除, 所以有

$$\frac{p < : \Pi}{\{\Pi \wedge N \wedge D\} \text{free}(p) \{ \Pi \wedge N \wedge (D \cup \text{equals}(p)) \}}.$$

由于我们并不需要借用指针逻辑中对非指针类型的分析结果, 因此非指针类型的赋值公理并不需要进行改动.

4 指针替换

通过上述修改后的指针逻辑规则对程序进行分析后, 我们可以得到完整地包含了指针从创建到最后销毁整个过程中所指向的对象信息的等价类集合, 利用该等价类集合中包含的信息, 我们就可以将目前代码属性分析不能处理的指针从源代码中替换掉而不影响源程序的行为. 考虑到目前的代码属性分析对非指针的处理已经比较完善, 我们考虑按定义 1 的格式用结构来表示指针类型. 然而, 在替换过程中指针和指针指向对象是不一样的, 定义 1 无法表示出对象和对象地址的差别, 因此我们修改定义如下所示.

定义 2. 对于 Ψ 中的指针 p , 都表示为一个四元组,

$$p = \langle \text{line}, \text{type}, \text{ap}, \text{loc} \rangle, \forall p < : \Psi,$$

其中,

(1) $\text{line} \in \text{Lines}$, 表示 p 当前在程序中的行号, Lines 为行号集合;

(2) $\text{type} \in \text{Types}$, 表示 p 指向的数据类型, Types 为 C 语言中的基本数据类型;

(3) $\text{ap} \in \text{AccessPaths}$, 表示 p 所对应的访问路径, AccessPaths 为访问路径集合;

(4) loc 为 32 位整数, 表示 p 所在的地址.

这样, 对于指针逻辑所能支持的指针操作, 都可以用上面的结构来描述, 替换规则如下.

(1) 指针赋值. 指针赋值传递的是地址, 同时左右值包含相同的访问路径, 因此对于 $p = q$ 有

$$p.\text{line} = \text{当前行号}; p.\text{obj} = q.\text{obj}; p.\text{loc} = q.\text{loc};$$

如果 q 为 Null, 则

$$p.\text{line} = \text{当前行号}; p.\text{loc} = q.\text{loc};$$

(2) 关系运算“=”和“!=”. 同样这里比较的

是地址, 因此, 对于 $p = q$ 和 $p != q$, 替换为

$$p.\text{loc} = q.\text{loc}; \text{ 和 } p.\text{loc} != q.\text{loc};$$

(3) 指针的脱引用. 对于所有“ $*p$ ”形式的引用都可以用 $p.\text{obj}$ 来替代, 对于结构指针来说, 替换会复杂一些: 如果结构域是对象, 如 $p \rightarrow \text{data} = \text{data}$ 则直接替换为

$$p.\text{obj}.\text{data} = \text{data};$$

如果结构域是指针, 如 $p \rightarrow \text{next} = q(\text{next})$ 和 q 均是指针), 那么替换规则为

$$p.\text{obj}.\text{next}.\text{loc} = q.\text{loc}; p.\text{obj}.\text{next}.\text{obj} = q.\text{obj}.$$

(4) $\text{free}(p)$. 此处将 $p.\text{obj}$ 置为空即可.

实际操作中, 指针 p 的 loc 域和 obj 域在指针分析时并不需要进行赋值, 可以在替换时随机地分配, 唯一的要求是不能有重复和不能为 Null, 因为 loc 为 Null 意味着 p 是空指针, obj 为 Null 则意味该指针指向的对象已经被释放. obj 的类型可以根据 type 的值来确定. 替换算法的伪代码如图 2 所示.

```

Input:  $\Pi, \text{Src}$ 
Output:  $\text{Src}$ 
1.  $P :=$  the equivalence class set of a pointer;
2.  $\text{UID}_P :=$  the unique variable name for  $P$ ;
3.  $\text{LOC} :=$  the address that a pointer point to;
4. while  $\Psi \neq \emptyset$  do
5.   SELECT  $P$  from  $\Psi$ ;
6.   GENERATE  $\text{UID}_P$  for  $P$ ;
7.   for each  $q$  in  $P$  do
8.     LOCATE the definition for  $q$  in  $\text{Src}$ ;
9.     if Successful then
10.      GENERATE  $\text{LOC}$  for  $q.\text{loc}$ ;
11.       $q.\text{obj} = \text{UID}_P$ ;
12.      REPLACE the line with  $q.\text{type}$  and  $\text{UID}_P$ ;
13.   end
14.   LOCATE  $q.\text{line}$  in  $\text{Src}$ ;
15.   if Successful then
16.     if  $\text{free}()$  in  $q.\text{line}$  then
17.       DELETE  $q.\text{line}$  in  $\text{Src}$ ;
18.     else
19.       if  $\text{malloc}()$  in  $q.\text{line}$  then
20.         GENERATE  $\text{LOC}$  for  $q.\text{loc}$ ;
21.          $q.\text{loc} = \text{LOC}$ ;
22.       else
23.         REPLACE  $q$  with  $\text{UID}_P$  in  $q.\text{line}$ ;
24.       end
25.   end
26. REMOVE  $P$  from  $\Psi$ ;
27. end
  
```

图 2 指针等价类替换算法

首先算法从 Ψ 选出一个指针等价类 P , 然后在源代码中遍历该等价类中的所有指针访问路径. 对于每个指针访问路径, 算法为其生成唯一的变量名 UID_P , 如果源代码中存在该指针的定义, 则用该指针逆向引用的数据类型定义 UID_P 来替换它; 如果指针所在行中存在 $\text{malloc}()$ 或 $\text{free}()$, 则删除该行, 因为属性验证对于这些函数调用的结果默认永远成

功;如果指针所在行对应该指针的访问路径的运算, 则用 *LOC* 替换, *LOC* 是为该指针等价类所指向对象生成的唯一的地址. 经过如是替换, 源代码中的指针已经全部转化为非指针类型的数据结构, 这样就可以作为属性验证的输入进行安全属性的验证. 如前所述, 属性验证通常假设无论任何时候 *malloc()* 和 *free()* 返回值都为 0, 所以替换中对 *malloc* 和 *free* 函数的改动并不会影响后面属性验证的分析结果.

5 实例及分析

为了说明本文分析方法的有效性, 我们同样采用文献[14]删除二叉排序树的节点的函数 *struct node *DeleteNode(struct node *p)* 为例, 现有的属性分析工具是无法对这样充满了各式指针引用的函数进行分析的. 由于篇幅所限, 原始程序欠奉, 请参见文献[14]中的图 1. 经过 3.2 节修改过的指针逻辑分析和我们提出的转换算法处理后, 得到的代码如图 3 所示.

```
1. struct node *DeleteNode(struct pointer p)
2. {
3.     struct pointer q, s;
4.
5.     if (p.obj.r.loc==NULL) {
6.         /* 右子树为空, 只需重接它的左子树 */
7.         q.obj=p.obj; q.loc=p.loc; q.lineno=6;
8.         s.obj=p.obj.l.obj; s.loc=p.obj.l.loc; s.lineno=7;
9.         free(q);
10.        return s;
11.    } else if (p.obj.l.loc==NULL) {
12.        /* 左子树为空, 只需重接它的右子树 */
13.        q.obj=p.obj; q.loc=p.loc; q.lineno=11;
14.        s.obj=p.obj.r.obj; s.loc=p.obj.r.loc; s.lineno=12;
15.        free(q);
16.        return s;
17.    } else { /* 左右子树均不空 */
18.        q.obj=p.obj; q.loc=p.loc; q.lineno=16;
19.        s.obj=p.obj.l.obj; s.loc=p.obj.l.loc; s.lineno=17;
20.        if (s.obj.r.loc==NULL) { /* 重接*q的左子树 */
21.            q.obj.l.obj=s.obj.l.obj; q.obj.l.loc=s.obj.l.loc;
22.            q.lineno=11;
23.            p.obj.data=s.obj.data;
24.            free(s);
25.            return p;
26.        } else {
27.            q.obj=s.obj; q.loc=s.loc; q.lineno=11;
28.            s.obj=s.obj.r.obj; s.loc=s.obj.r.loc; s.lineno=;
29.            while (s.obj.r.loc != NULL) {
30.                /* 转左, 然后向右右前进到尽头 */
31.                q.obj=s.obj; q.loc=s.loc; q.lineno=11;
32.                s.obj=s.obj.r.obj; s.loc=s.obj.r.loc; s.lineno=;
33.            }
34.            p.obj.data=s.obj.data;
35.            q.obj.r.obj=s.obj.l.obj; q.obj.r.loc=s.obj.l.loc;
36.            q.obj.r.line=; /* 重接*q的右子树 */
37.            free(s);
38.            return p;
39.        }
40.    }
```

图 3 指针替换后的二叉树节点删除代码

对比源程序和上图可以看出, 所有的指针都已经替换为非指针类型, 而所有的指针指向信息都可以完整地反映出来. 这样就可以供 BOOP 或者类似的属性验证工具进行普通的可达路径检查了.

6 结论及下一步工作计划

安全操作系统的实现代码和安全属性之间的误差一直是人们希望消除的对象, 虽然 C 语言指针导致的安全隐患一直为研究人员所诟病, 但其可媲美低级语言的执行高效率使得近年来兴起的类型安全语言, 如 ML、Java 等相形见绌, 尤其在操作系统这样的效率优先的应用中, C 语言更具备不可替代性. 然而, 传统的手工劳动已经不适用于现代软件开发, 尤其是像操作系统这样的大规模应用. 而静态代码属性验证工具对指针的支持也一直很不理想.

本节在分析和总结前人工作的基础上, 提出了一种改进的代码安全属性验证方法. 该方法在利用传统的源代码安全属性验证工具的基础上, 加入指针逻辑, 该逻辑综合了基于类型和基于逻辑两种主程序分析方法的优点, 可以实现对指针程序的精确分析. 和文献[14]试图将这种指针逻辑开发一种原形语言, 并针对该语言设计专门的编译器在编译阶段对代码的安全性进行证明不同, 我们直接利用指针逻辑对源代码的分析结果对源代码中的指针进行替换, 从而避开了传统静态代码属性验证工具对指针处理功能太弱的瓶颈, 可以实现对 C 语言中的部分指针及运算进行处理, 其结果既可应用于一定的实际代码, 也保持了较低的误报率.

由于我们的方法主要依赖于指针逻辑生成的指针等价类, 而指针逻辑原本是配合 PointerC 使用的, 该语言对标准 C 中的指针用法作了一定的限定. 王志芳等人^[16]在文献[14]的基础上对指针逻辑进行了扩展以纳入指针运算和指针函数调用, 因此, 如何将指针运算和指针函数的处理融入我们的方法则是下一步工作的主要任务.

参 考 文 献

[1] Clarke Edmund M, Grumberg Orna, Peled Doron A. Model Checking. Massachusetts: The MIT Press, 1999

[2] Nipkow T, Paulson L. Isabelle/HOL—A proof assistant for higher-order logic//Lecture Notes in Computer Science 2283. Springer, 2008

- [3] Cruz Jorge. Constraint Reasoning for Differential Models. Amsterdam: The IOS Press, 2005
- [4] Weißenbacher Georg. An abstraction/refinement scheme for model checking C programs [M. S. dissertation]. Technischen Universität Graz, Graz, Austria, 2003
- [5] Henzinger T A, Jhala R, Majumdar R, Necula G C, Sutre G, Weimer W. Temporal-safety proofs for systems code//Proceedings of the 14th International Conference on Computer-Aided Verification (CAV). Copenhagen, Denmark, 2002: 526-538
- [6] Clarke M, Kroening D, Sharygina N, Yorav K. SATABS: SAT-based predicate abstraction for ANSI-C//Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005). Edinburgh, U. K., 2005: 570-574
- [7] Flanagan C, Leino K R M, Lillibridge M, Nelson G, Saxe J B, Stata R. Extended static checking for java//Proceedings of the ACM SIGPLAN2002 Conference on Programming Language Design and Implementation (PLDI'2002). Berlin, Germany, 2002: 234-245
- [8] Barnett M, Leino K R M, Schulte W. The Spec# programming system: An overview//Proceedings of the CASSIS'04. Marseille, France, 2005: 49-69
- [9] Filliatre J-C, Marche C. Multi-prover verification of C programs//Proceedings of the ICFEM. Seattle, USA, 2004: 15-29
- [10] Blazy Sandrine, Dargaye Zaynah, Leroy Xavier. Formal verification of a C compiler front-end//Proceedings of the Symposium on Formal Methods (FM'06). Hamilton, Canada, 2006: 460-475
- [11] Hua Bao-Jian, Chen Yi-Yun, Li Zhao-Peng, Wang Zhi-Fang, Ge Lin. Design and proof of a safe programming language PointerC. Chinese Journal of Computers, 2008, 31(4): 556-564(in Chinese)
(华保健, 陈意云, 李兆鹏, 王志芳, 葛琳. 安全语言 PointerC 的设计及形式证明. 计算机学报, 2008, 31(4): 556-564)
- [12] Necula G C, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code//Proceedings of the 29th ACM Symposium on Principles of Programming Languages, 2002: 128-139
- [13] Jim T, Morrisett J G, Grossman D, Hicks M V, Cheney J, Wang Y. Cyclone: A safe dialect of C//Proceedings of the General Track: 2002 USENIX Annual Technical Conference. Berkeley; USENIX Association. Monterey, CA, USA, 2002: 275-288
- [14] Chen Yi-Yun, Hua Bao-Jian, Ge Lin, WANG Zhi-Fang. A pointer logic for safety verification of pointer programs. Chinese Journal of Computers, 2008, 31(3): 372-380(in Chinese)
(陈意云, 华保健, 葛琳, 王志芳. 一种用于指针程序安全性证明的指针逻辑. 计算机学报, 2008, 31(3): 372-380)
- [15] Xi H W. Applied type system(extended abstract)//Post-Workshop Proceedings of the TYPES2003. Lecture Notes in Computer Science 3085. Berlin: Springer-Verlag, 2004: 394-408
- [16] Wang Zhi-Fang, Chen Yi-Yun, Wang Zhen-Ming, Wang Wei, Tian Bo. An extension to pointer logic for verification//Proceedings of the 2nd IEEE IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2008). Nanjing, China, 2008: 49-56



ZHANG Yang, born in 1971, Ph.D.. Her research interests focus on secure operating system and its verification.

CHENG Liang, born in 1982, Ph. D. candidate. His research interests include secure operating system and its verification.

Background

The work attributes to the project “Security Analysis of Windows Vista Kernel and Operating System Controllability Research”, which is supported by National High Technology Research and Development Program (863 Program) of China under grant Nos. 2007AA01Z465, 2006AA01Z433, 2007AA01Z414.

The code security property verification is an effective way to detect the vulnerability of operating system, which draws wide attention nowadays. Although current code security property verification tools can check various properties of a C program such as whether a certain point is reachable, or whether there exist race conditions in the program, the anal-

ysis difficulties brought in by the flexible pointer usage limits the scope of these tools' application to the real world.

In this paper, the authors propose an improved verification method for code security property inspired by the pointer logic. Firstly the program be traversed to be checked with the pointer logic, in order to obtain the equivalence classes of all the pointers. Then, the replacement algorithm is given to convert the pointers in the program to non-pointer type according to the information of the equivalence classes. Without the fuzziness introduced by pointers, the traditional code security property verification tools can do all their property check to the real world code thoroughly.