一种面向堆操作程序内存安全性的域敏感内存模型

董龙明 王 戟 陈立前 刘江潮

(国防科技大学计算机学院并行与分布处理国家重点实验室 长沙 410073)

摘 要 堆操作程序具有通过共享易变数据结构动态操纵堆内存单元的特性,使得内存安全性难以保证。针对这个问题,提出了一种域敏感的k-limit内存抽象模型,以支持动态调整抽象的粒度,取得静态分析在精度和效率上的平衡。分别从框架、性质、操作方面介绍了该内存模型,然后结合内存安全性的定义,在基于该模型的操作语义框架内定义了4种与内存安全性相关的错误类型,最后设计了基于该模型内存安全性检测的数据流迭代算法。

关键词 堆操作程序,内存安全性,k-limit内存抽象模型,动态可调节

中图法分类号 TP301 **文献标识码** A

Field-sensitive Memory Model for Memory Safety of Heap-manipulating Programs

DONG Long-ming WANG Ji CHEN Li-qian LIU Jiang-chao

(National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract Heap-manipulating programs usually operate memory cells directly through shared and mutable data-structures, which makes their memory safety more complex and harder to guarantee. A field-sensitive k-limit abstract memory model was proposed in this paper to support dynamic adjustment of the precision and efficiency of the analysis. We presented its framework, property and operations. And then, four kinds of memory-related errors were identified in the operational semantics of the abstract memory model according to the definition of memory safety. In the end, we proposed the dataflow iteration algorithm for detecting the memory safety of C programs.

Keywords Heap-manipulating programs, Memory safety, k-limit abstract memory model, Dynamic adjustment

1 引言

堆操作程序是一种使用常用数据结构(如链表、树等)来直接操作内存单元的程序,在这类程序运行过程中往往伴随着平凡的内存单元分配、释放、合并、分离等操作,因此,这些程序的堆内存表现为共享、易变等动态特性。保证堆操作程序的内存安全性比其他程序困难和复杂得多,一方面,需要考虑各类指针值是否有效(即指向的内存单元是否被释放,无效指针解引用错误);另一方面,需要考虑内存单元间的链接关系,是否存在孤立内存单元(内存泄露错误)。堆操作程序的内存状态可以使用一个二元组$\langle H, S\rangle$表示,其中:$H=\langle V, E\rangle$,结点V表示所有堆内存单元集,$E: V \xrightarrow{F} V$表示所有结点间通过指针域F构成的链接关系;$S: PVar \rightarrow V$,$PVar$表示程序中指针变量集合,描述了指针变量的值为内存的地址。图1为一个简单的内存状态,分别使用二元组和指向图表示。

$$PVar = \{x, y, z\}$$
$$S = \{\langle x, heap\_0\rangle, \langle y, heap\_1\rangle, \langle z, heap\_4\rangle\}$$
$$V = \{heap\_0, heap\_1, heap\_2, heap\_3, heap\_4\}$$
$$E = \{heap\_0 \xrightarrow{f_1} heap\_3, heap\_0 \xrightarrow{f_2} heap\_1, heap\_1 \xrightarrow{f_2} heap\_2, heap\_3 \xrightarrow{f_1} heap\_4\}$$

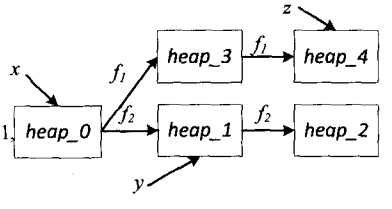

图1 一个简单的堆内存状态

内存安全性是程序安全性的一种重要性质,指程序运行过程中不会出现与内存相关的错误,常见的内存错误包括:内存泄露、空指针引用、悬挂指针引用、多次释放。这些内存相关错误在软件所有的错误中比例较大,Chou等人[1]分析和统计Linux和OpenBSD内核错误时,共1025个错误中有600个内存相关的错误,约占58.5%。这类错误能够导致重要数据泄露,或者长期运行着的系统性能下降或由于内存消耗完而崩溃。

当前,内存安全性分析有多种方法及其工具,主要分为动态和静态的方法[2]。典型动态方法的工具有Purify[3]、JPF[4]、Valgrind Memcheck[5]等,这类方法虽能准确地定位错误,自动化程度高,但受限于输入用例,不能检查所有内存安

到稿日期:2011-11-02 返修日期:2012-01-14 本文受国家自然科学基金(61120106006,90818024)资助。

董龙明(1982—),男,博士生,主要研究方向为堆操作程序分析与验证,E-mail:donglongming22@163.com。

全性相关的错误；静态方法在不运行软件前提下检查是否存在内存安全性威胁，典型工作有：LCLint[6]、SATURN[7]、SoftBound[8]等，但是存在较高的误报率。因此，如何在保证精度的前提下，使静态分析工具能够分析大规模程序是当前静态分析研究的热点和难点。检测内存安全性相关的各种类型错误，需要为程序设计统一的堆内存模型来支持程序语句的状态迁移，并基于该内存模型定义和识别各种内存相关的错误类型。结合堆操作程序的特点，面向堆操作程序内存安全性的堆内存模型是域敏感的，能够识别不同指针域上的操作。

数据流迭代算法是一种经典的静态分析算法[9]，在程序控制流图上寻找关于数据流事实的不动点，在迭代过程中检测各种程序相关的错误。堆操作程序具有共享易变的内存特点，完全的内存状态是无穷的，因此，数据流迭代算法需要保证其终止性。研究堆操作程序内存安全性时，我们在文献[10,11]中分别针对内存泄露和指针非法解引用错误，设计了$k=1$的内存抽象模型来保证迭代算法的终止性。一系列基准程序的实验表明：该方法能够在分析精度和效率上取得了较好的平衡。在此基础上，提出了一种带参数的k-limit内存抽象模型，研究了该内存模型的复杂度，并且设计了支持精度动态可调的操作。

本文的主要创新点和贡献有：

1）提出了一种带参数的k-limit内存抽象模型。它是域敏感的，能够描述指针变量间的别名和局部可达的关系。

2）设计了支持分析精度动态可调的操作。分别设计内存模型的正规抽象和精化操作，并且针对模型的复杂度设计了面向不同内存安全性的简化模型抽象和精化操作。

3）提出了面向内存安全性数据流迭代检测算法。基于该k-limit模型，定义堆操作程序基本语句的操作语义和各种内存安全性相关的错误，最后，提出了一种统一的检测内存安全性数据流迭代算法框架。

2 域敏感的堆内存模型

2.1 堆操作程序

堆操作程序常常使用结构体(C语言中的struct类型)表示内存中一个聚集单元，结构类型中指针域用来链接指向另一个聚集单元。由于数据域上的操作不会引发内存相关的错误，为表示简洁，本文省略数据域上的操作。图2给出了堆操作程序的最小语法集。

$$
\begin{aligned}
&p,q \in PVar\\
&f_1,f_2,\ldots,f_i \in Fields\\
&A\,\mathrm{sgn}\ Stmnt := p = null \mid p \to f_i = null \mid p = q \mid p = q \to f_i \mid\\
&\qquad\qquad p \to f_i = q \mid p = malloc() \mid p = free()\\
&SequenceStmnt := A\,\mathrm{sgn}\ Stmnt;\ A\,\mathrm{sgn}\ Stmnt\\
&SwitchStmnt := switch\quad e\quad (c_1 : n_1,\ldots,c_k : n_k;\ldots)\\
&CallStmnt := e = f(e_1,\ldots,e_k)\\
&\mathrm{Re}\,turnStmnt := return\quad e
\end{aligned}
$$

图2 堆操作程序最小语法集

图2中，$PVar$表示堆操作程序中指针变量集，基本指针赋值语句包括：将一个指针变量(或指针域)置为空，将一个指针变量(或指针域)的值拷贝到另一个指针(或指针域)，并申请或释放内存单元。堆操作程序支持过程间函数调用和返回，堆操作程序中的控制结构(如If-Then-Else和While)可以转换为基本switch条件选择语句，其中e表示测试条件，c_i是每个分支成立的常量，n_i表示相应的后继结点，例如当条件$e=c_i$成立时，则控制流向结点n_i。

任何C程序都可以通过切片(slicing)和转换预处理过程转换为精简的堆操作程序。首先，将那些不影响指针变量值和内存拓扑结构的赋值语句从程序中删除；然后，引入辅助指针变量(instrumental pointer variables)，将堆操作程序转换成图2定义的标准形式，转换规则见表1。

表1 堆操作程序的转换规则

基本指针赋值语句	转换后的语句
$p\to f_i=q\to f_j$	$pt_0=q\to f_j;p\to f_i=pt_0$
$p=p\to f_i$	$pt_1=p\to f_i;p=pt_1$
$p\to f_i=\mathrm{malloc}$	$pt_2=\mathrm{malloc};p\to f_i=pt_2$
$p=q\to f_i\to f_j$	$pt_3=q\to f_i;p=pt_3\to f_j$
$p\to f_i=\mathrm{free}()$	$pt_4=p\to f_i;pt_4=\mathrm{free}()$

2.2 k-limit 内存抽象模型

分析堆操作程序基本语句可知：在一条指针赋值语句中，指针变量直接(或间接)引用的内存结点距离该指针变量的值是有限的。例如：语句$p\to n=q$中，将指针变量p指向结点的指针域n的值修改为指针q指向的内存结点的地址，在此语句中，指针p通过指针域n引用的内存结点距离指针p的值为1。因此，堆操作程序中指针变量引用的内存单元具有局部性。

堆操作程序中，一个具有n个指针域聚集结构体类型的指针变量可以表示为$p:\{f_1,f_2,\cdots,f_n\}$，其中$f_1,f_2,\cdots,f_n$为该结构体中的指针成员，各指针成员的值为聚集内存单元的首地址。在堆内存中，由于指针(或指针域)间的赋值操作，堆内存单元相互链接构成各种形态(shape)，比如常用的数据结构、链表或树等。具有聚集结构体类型的指针可以通过解引用(dereferecing)其指针成员来遍历所有可达的内存单元或修改其中的链接关系，比如一个指向链表头结点的指针通过next指针域依次遍历所有链表的结点，或者插入某个内存单元。因此，指针变量通过其指针成员解引用操作，能够得到所指向的堆内存拓扑结构。指针别名(alias)描述了两个或多个指针的值相等关系，即指向同一内存单元，对检测内存安全性极为重要，比如检测修改某个指针变量的值是否引起内存泄露错误时，需要考虑当前状态下是否有其他指针与该指针变量别名；检测释放某个内存单元是否引起悬挂指针解引用(dangling dereferencing)时，需要考虑该内存单元是否被其他指针所指向。因此，一个支持检测堆操作程序内存安全性的堆内存模型需要考虑两方面的因素：1)域敏感的，区分各种通过聚集结构体中各指针成员解引用的路由路径；2)别名信息，描述所有通过解引用操作可达内存单元上的指针别名集。

由于通过指针域解引用的路由路径是无穷的，因此需要进行抽象。基于分析精度和效率的考虑，本文提出了一种可扩展的k-limit内存抽象模型，如图3所示。k表示内存中通过指针域解引用操作到达的内存结点距离指针所指向内存节点的距离，例如$k=0$表示指针p所指向的内存单元；ASet表示指针别名集(Alias Set)，是指针变量构成集合的幂集2^{PVar}的子集。为了精确地刻画内存安全性相关的特性，ASet中有3个特殊的元素：$\emptyset$表示程序中没有其他指针变量指向该内存单元；$\perp$表示指针(或指针域)的值为null，堆内存中该内存单元还没有分配；Θ表示某个指针(或指针域)的值是无效内存单元(即该内存单元已经通过其他指针被释放)，这样的指针也叫做悬挂指针(dangling pointer)。因此，指针别名集ASet的最大个数为：$2^{|PVar|}+2$。从堆内存拓扑结构角度考虑，k-limit内存抽象模型能够描述距离指针所指向的内存单元值在k以内的精确指针别名信息，当距离大于k时，需要进

行抽象。可以由一个二元结构 $K=\langle\{0,1,2,\cdots,k,k+1\},\{+,-\}\rangle$表示，操作定义如下：

$$k_1+k_2=\begin{cases}k_1+k_2, & 若\ k_1+k_2<k+1\\ k+1, & 若\ k_1+k_2\geqslant k+1\end{cases}$$

$$k_1-k_2=\begin{cases}\perp, & 若\ k_1<k_2\\ k+1,k,k-1,\cdots,k+1-k_2, & 若\ k_1=k+1\\ k_1-k_2, & 其他\end{cases}$$

式中，表达式 k_1-k_2（或者 k_1+k_2）描述了从当前距离指针变量值为 k_1 的内存结点出发，沿着指针域解引用路由向后（或前）经过 k_2 步后得到距离该指针变量的值。元素 $k+1$ 是一个抽象值，描述了内存中所有距离当前内存结点值等于或大于 $k+1$ 的内存结点（又称摘要结点，summary node）；元素 $\perp$ 表示没有定义，该运算符不能够作用在此操作数上。因此，基于 k-limit 内存抽象框架，可以定义一种指针变量扩展结构，描述指针间域敏感的别名关系。k-limit 内存抽象框架中，指针通过其指针成员解引用到达堆内存结点，比如到达距离指针变量 p 值为 1 的内存结点，可能的路由路径包括：$p\rightarrow f_1$，$p\rightarrow f_2$，$\cdots$，$p\rightarrow f_n$，共 n 种情况，使用集合 R_k 表示所有到达距离指针变量值为 k 的内存结点的解引用路由路径，集合 R 表示所有 $R_k(k=0,1,2,3,\cdots)$构成的集合。所有指针解引用的路由路径个数与解引用深度 k 的关系见表 2。假设聚集结构体中指针成员最大个数为 n。

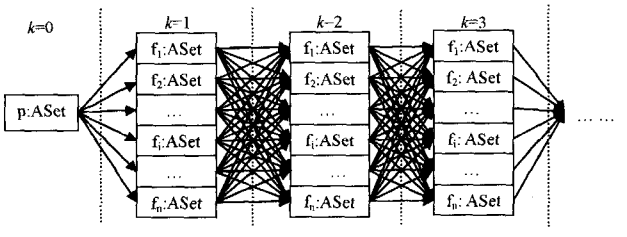

图 3　k-limit 内存抽象模型

表 2　k-limit 内存抽象框架解引用路由路径个数

解引用深度 k 值	解引用路由路径个数 $\lvert R_k\rvert$
0	1
1	n^1
2	n^2
3	n^1
…	…
k_i	n^{ki}
…	…

如果解引用深度为 k，所有可能的解引用路由的路径个数为：$|R|=1+n^1+n^2+n^3+\cdots+n^k=\frac{n^{k+1}-1}{n-1}(n>1)$，$|R|=1+k(n=1)$。

定义 1（扩展的指针结构，extended pointer structure）　k-limit 内存抽象框架下，描述指针别名关系域敏感的扩展结构 $\tau_p^{\#}$ 可以表示为：$\{0:\langle R_0;2^{PVar}\rangle;1:\langle R_1;2^{PVar}\rangle;\cdots;k_i:\langle R_{ki};2^{Pvar}\rangle;\cdots;k:\langle R_k;2PVar\rangle;k+1:\langle R_k^*;2^{PVar}\rangle;\}$。其中 $0,1,\cdots,k_i,\cdots,k$ 为距离当前指针的精确值；$k+1$ 是一个抽象值，表示距离当前指针等于或大于 k 的所有值；$R_0,R_1,\cdots,R_{ki},\cdots,R_k$ 为所有可能解引用的路由路径；R_k^* 为所有 R_k 上的路由路径扩展，表示一类前 k 位为 R_k 的路由路径。

根据上述指针扩展结构的定义，可以得到图 1 中指针 x、y 和 z 的扩展结构，可以表示为：

$\tau_x^{\#}:\{0:\langle x,\emptyset\rangle;1:\langle x\rightarrow f_1,\emptyset\rangle\langle x\rightarrow f_2,\{y\}\rangle;2:\langle x\rightarrow f_1\rightarrow f_1,\{z\}\rangle\langle x\rightarrow f_1\rightarrow f_2,\perp\rangle\langle x\rightarrow f_2\rightarrow f_1,\perp\rangle\langle x\rightarrow f_2\rightarrow f_2,\emptyset\rangle;3:\langle R_3,\perp\rangle\}$

$\tau_y^{\#}:\{0:\langle y,\emptyset\rangle;1:\langle y\rightarrow f_1,\perp\rangle\langle y\rightarrow f_2,\emptyset\rangle;2:\langle R_2,\perp\rangle\}$

$\tau_z^{\#}:\{0:\langle z,\emptyset\rangle;1:\langle R_1,\perp\rangle\}$

为了得到指向某个指针变量，通过给定解引用路径到达的内存单元的所有指针变量集，定义函数 $KR_\tau:K\times R\rightarrow 2^{PVar}$。例如：图 1 中，$KR_{\tau_x^{\#}}(1,x\rightarrow f_2)=\{y\}$。

定义 2（指针扩展结构包含关系，Containment）　指针变量 x 的两种扩展结构分别为 $\tau_{1x}^{\#}$ 和 $\tau_{2x}^{\#}$，如果 $\forall k\in K\,\forall r_k\in R_k.\ KR_{\tau_{1x}^{\#}}(k,r_k)\subseteq KR_{\tau_{2x}^{\#}}(k,r_k)$，那么 $\tau_{1x}^{\#}$ 包含于 $\tau_{2x}^{\#}$，记作 $\tau_{1x}^{\#}\subseteq\tau_{2x}^{\#}$。如果 $\forall k\in K\,\forall r_k\in R_k.\ KR_{\tau_{1x}^{\#}}(k,r_k)\subset KR_{\tau_{2x}^{\#}}(k,r_k)$，那么 $\tau_{1x}^{\#}$ 真包含于 $\tau_{2x}^{\#}$。

由此可知，指针的两个扩展结构是相等的，可以描述为：$\tau_{1x}^{\#}=\tau_{2x}^{\#}\equiv\tau_{1x}^{\#}\subseteq\tau_{2x}^{\#}\wedge\tau_{1x}^{\#}\supseteq\tau_{2x}^{\#}$。

定义 3（指针扩展结构兼容关系，Compatibility）　指针变量 x 两种扩展结构 $\tau_{1x}^{\#}$ 和 $\tau_{2x}^{\#}$ 是兼容的，当且仅当 $\tau_{1x}^{\#}\subseteq\tau_{2x}^{\#}\vee\tau_{1x}^{\#}\supseteq\tau_{2x}^{\#}$。

定义 4（堆内存局部抽象表示）　k-limit 内存抽象框架下，堆操作程序 HP 中，任意程序点处局部内存状态 $S^{\#}$ 可以由所有指针变量的扩展结构来表示，即 $S^{\#}=\{\tau_{p_i}^{\#}\mid p_i\in Live\text{-}PVar(HP)\}$。

k-limit 内存抽象框架下，基于指针扩展结构的抽象状态数是有限的，最大数为：$[\frac{2n^{k+1}-n^k-1}{n-1}\times(2^{pn}+2)]^{pn}(n>1)$，其中，$pn$ 表示程序中指针变量的个数，n 为聚集结构体中具有指针类型成员变量的个数，$2^{pn}+2$ 是可能的指针别名集的个数。

由于指针间的别名关系，一个抽象堆状态可能具有多种形式。为了得到堆内存间的抽象状态间的关系，需要将抽象状态转换为标准形式（canonical form），也叫做饱和状态。

定义 5（堆内存局部抽象饱和状态，saturated）　对任意抽象状态 $S^{\#}$，如果满足以下 3 条性质，那么 $S^{\#}$ 是饱和的：

1）反自反性，$\forall x\in PVar.\ x\notin KR_{\tau_x^{\#}}(0,x)$；

2）对称性，$\forall x,y\in PVar.\ y\in KR_{\tau_x^{\#}}(0,x)\rightarrow x\in KR_{\tau_y^{\#}}(0,y)$；

3）传递性，$\forall x,y,z\in PVar.\ y\in KR_{\tau_x^{\#}}(k_1,R_{k1})\wedge z\in KR_{\tau_y^{\#}}(k_2,R_{k2})\rightarrow z\in KR_{\tau_x^{\#}}(k_1+k_2,R_{k1}+R_{k2})$。

```
Algorithm Saturate(S#)
  modified←true;
While modified do
  modified←false;
  For each τx# in S#
    //Anti-reflexivity
    KRτx#(0,x)←KRτx#(0,x)-{x};
    For y∈ KRτx#(0,x)
      //Symmetry
      If x∉KRτy#(0,y) then
        modified←true;
        KRτy#(0,y)←KRτy#(0,y)∪{x};
      For y∈KRτx#(k1,Rk1)
        //Transitivity
        If z∈KRτy#(k2,Rk2) ∧ z∉KRτx#(k1+k2,Rk1+Rk2)
        modified←true;
        KRτx#(k1+k2,Rk1+Rk2)←KRτx#(k1+k2,Rk1+Rk2)∪{z}
      End For
    End For
  End For
End While
```

图 4　饱和操作

条件 3)中,公式 $R_{k1}+R_{k2}$ 表示解引用路由有 k_1+k_2 位,且前 k_1 位为路径 R_{k1},后 k_2 位为 R_{k2}。任意一个堆内存局部抽象状态可以通过饱和操作(saturate)达到饱和状态,如图 4 所示。

与之对应,为了节省抽象状态的存储开销,可以定义堆内存抽象精简状态(compactable)。它是一种没有任何冗余信息的状态,需要满足以下 3 个条件:

1) 反自反性,$\forall x\in PVar.\ x\notin KR_{\tau_x}^{\#}(0,x)$;

2) 反对称性,$\forall x,y\in PVar.\ y\in KR_{\tau_x}^{\#}(0,x)\wedge x\in KR_{\tau_y}^{\#}(0,y)=\text{false}$;

3) 反传递性,$\forall x,y,z\in PVar.\ y\in KR_{\tau_x}^{\#}(k_1,R_{k1})\wedge z\in KR_{\tau_y}^{\#}(k_2,R_{k2})\rightarrow z\notin KR_{\tau_x}^{\#}(k_1+k_2,R_{k1}+R_{k2})$。

任意一个堆内存局部抽象状态都可以通过精简操作(compact)达到精简状态,如图 5 所示。

```
Algorithm Compact(S#)
  modified ← true;
While modified do
  modified ← false;
  For each τx# in S#
    //Anti-reflexivity
    KRτy#(0,x) ← KRτx#(0,x) - {x};
    For y ∈ KRτx#(0,x)
      //Anti-symmetry
      If x ∈ KRτy#(0,y) then
        modified ← true;
        KRτy#(0,y) ← KRτy#(0,y) - {x};
      For y ∈ KRτx#(k1,Rk1)
        //Anti-transitivity
        If z ∈ KRτy#(k2,Rk2) ∧ z ∈ KRτx#(k1+k2,Rk1+Rk2)
          modified ← true;
          KRτy#(k1+k2,Rk1+Rk2) ← KRτx#(k1+k2,Rk1+Rk2) - {z};
      End For
    End For
  End For
End While
```

图 5 精简操作

由于程序固有属性(控制流汇聚和数据流迭代算法)多次迭代,因此常常需要合并多个堆内存局部抽象状态。一方面能减少存储开销,加速迭代算法的终止;另一方面由于合并操作带来的不精确性,导致程序分析精度的损失。因此,为了尽可能地检测出堆操作程序内存安全性的错误,合并操作具有两种方式,如图 6 所示,当且仅当两个抽象状态存在包括关系时才能进行合并。假设需要合并的两个堆抽象状态为 $S_1^{\#}$ 和 $S_2^{\#}$。

$$Join_1(S_1^{\#},S_2^{\#})=\begin{cases}S_1^{\#}, & \text{if } S_1^{\#}\subseteq S_2^{\#}\\ S_2^{\#}, & \text{if } S_1^{\#}\supseteq S_2^{\#}\\ S_1^{\#}\cup S_2^{\#}, & \text{otherwise}\end{cases}$$

$$Join_2(S_1^{\#},S_2^{\#})=\begin{cases}S_2^{\#}, & \text{if } S_1^{\#}\subseteq S_2^{\#}\\ S_1^{\#}, & \text{if } S_1^{\#}\supseteq S_2^{\#}\\ S_1^{\#}\cup S_2^{\#}, & \text{otherwise}\end{cases}$$

图 6 Join 操作

根据抽象状态饱和定义可知,如果 $S_1^{\#}\subseteq S_2^{\#}$,对每个指针变量,$S_2^{\#}$ 在通过相同指针域解引用路由路径得到的内存结点处比 $S_1^{\#}$ 具有更多的别名信息,即在抽象状态 $S_1^{\#}$ 下,如果被某个指针指向或通过指针域解引用的某个内存单元被另一个指针指向,那么在状态 $S_2^{\#}$ 一定也是如此。

当检测内存泄露错误时,如果某条语句在抽象状态 $S_2^{\#}$ 下也会发生内存泄露错误,那么在状态 $S_1^{\#}$ 下一定会发生同样的内存泄露错误;但是,如果某条语句在状态 $S_1^{\#}$ 下发生内存泄露错误,同样的内存泄露错误不一定会在状态 $S_2^{\#}$ 下发生,因为状态 $S_1^{\#}$ 下泄露的单元可能在状态 $S_2^{\#}$ 下被其他指针引用。所以,在检测内存泄露时,合并需要采用 $Join_1$ 操作策略。

当检测堆操作程序指针非法解引用时,如果 $S_1^{\#}\subseteq S_2^{\#}$,那么,在抽象状态 $S_1^{\#}$ 下某条语句发生非法指针解引用错误时,相同的错误同样在状态 $S_2^{\#}$ 下一定会发生;另一方面,如果在状态 $S_2^{\#}$ 下某条语句发生非法指针解引用错误,而相同的非法指针解引用错误不一定在状态 $S_1^{\#}$ 下发生,则状态 $S_2^{\#}$ 下指向已经释放的内存单元的指针在状态 $S_1^{\#}$ 不一定还指向该释放的内存单元。因此,当检测非法指针解引用错误时,合并操作需要采用 $Join_2$ 操作策略。

2.3 动态可调操作

k-limit 内存抽象模型下,分析和检测内存安全性时,首先根据程序的特点和规模,初始化设置一个 k 值建立内存抽象模型,k 值既不能太大也不能太小。根据抽象模型复杂度公式可知,随着 k 值的增加,公式$\left[\frac{2n^{k+1}-n^k-1}{n-1}\times(2^m+2)\right]^m$的复杂度呈指数增加,如果设置的 k 值比较大,不但会消耗大量的内存,而且会使迭代次数增多,在有限的时间内难以终止,使得分析的效率比较低;另一方面,k 值如果比较小,内存模型中与指针距离大于 k 部分的内存单元被截断抽象成一个摘要结点,可能产生比较多的误报,这就会影响分析的精度。一个良好的、可伸缩的静态分析技术能够支持动态调整 k 值的大小:初始分析时,设置一个比较小的 k 值;当分析产生比较多的误报时,增加 k 值精化内存模型,以提高分析的精度;当分析占用过多的内存和 cpu 资源时,减少 k 值抽象内存模型,以提高分析的效率。动态调节过程需要两个操作:正规抽象(Canonical abstraction)和模型精化(refinement)。

定义 6(正规抽象 Abs) k_1-limit 框架下指针 p 的扩展结构可以表示为:$\tau_p^{\#}=\{0:\langle R_0;2PVar\rangle;1:\langle R_1;2^{PVar}\rangle;\cdots;k_i:\langle R_{ki};2^{Pvar}\rangle;\cdots;k_1:\langle R_{k1};2P^{var}\rangle;k_1+1:\langle R_{k1}^*;2^{PVar}\rangle;\}$,正规抽象后,在 k_2-limit 框架下扩展结构 $Abs(\tau_p^{\#})$,其中 $k_2<k_1$。转换过程如下:

(1) $KR_{Abs(\tau_p^{\#})}(k,R_k)=KR_{\tau_p^{\#}}(k,R_k)$,若 $k\leqslant k_2$;

(2) $KR_{Abs(\tau_p^{\#})}(k_2+1,R_{k2}^*)=KR_{\tau_p^{\#}}(k_2+1,R_{k2+1})\cup\cdots\cup KR_{\tau_p^{\#}}(k_1,R_{k1})\cup KR_{\tau_p^{\#}}(k_1,R_{k1}^*)$。

正规抽象将精确模型转换为更加抽象的模型时,保持指针周围的内存单元别名信息不变,将大于给定 k 值的内存结点抽象合并为一个摘要结点。堆操作程序指针赋值语句中,解引用指针操作具有局部性,优先遍历指针变量周围的内存单元,因此,正规抽象能够在抽象过程中尽可能地减少误报。模型精化过程是将摘要结点展开成更多结点,按照 k 值逐步精化模型,因此,该过程是演绎的。

定义 7(模型精化 Ref) k-limit 框架下指针 p 的扩展结构表示为:$\tau_p^{\#}=\{0:\langle R_0;2^{PVar}\rangle;1:\langle R_1;2^{PVar}\rangle;\cdots;k_i:\langle R_{ki};2^{Pvar}\rangle;\cdots;k:\langle R_k;2^{PVar}\rangle;k+1:\langle R_k^*;2^{PVar}\rangle;\}$,模型精化后,在 $k+1$ 框架下,扩展结构 $Ref_1(\tau_p^{\#})$,假设聚集结构类型的指针成员为:$f_1,f_2,\cdots,f_n$,转换过程如下:

(1) $KR_{Ref_1(\tau_p^\#)}(k_i,R_{ki})=KR_{\tau_p^\#}(k_i,R_{ki})$若 $k_i\leqslant k$;

(2) $KR_{Ref_1(\tau_p^\#)}(k+1,R_{k+1})\cup KR_{Ref_1(\tau_p^\#)}(k+2,R_{k+1}^*)=KR_{\tau_p^\#}(k+1,R_k^*)$。

k-limit 框架下每个摘要结点域敏感地分裂成 n 个内存单元和 n 个新的摘要结点，且结点上的指针别名集满足(2)。假设精化前摘要结点的指针别名集中元素个数为 m，则精化可以产生 $(2n)^m$ 种情况；精化后的状态空间比较大，将影响分析的效率，面向需要验证特定的性质，在不产生漏报的情况下，可以设计特定的简化模型来精化策略。

定义 8(简化的内存泄露模型精化$\hat{\text{RefL}}$) k-limit 框架下指针 p 的扩展结构表示为：$\tau_p^\# =\{0:\langle R_0;2^{PVar}\rangle;1:\langle R_1;2^{PVar}\rangle;\cdots;k_i:\langle R_{ki};2^{PVar}\rangle;\cdots;k:\langle R_k;2P^{var}\rangle;k+1:\langle R_k^*;2^{PVar}\rangle;\}$，模型精化后，在 $k+1$ 框架下扩展结构$\hat{RefL}_1(\tau_p^\#)$，假设聚集结构类型的指针成员为：$f_1,f_2,\cdots,f_n$，转换过程如下：

(1) $KR_{\hat{RefL}_1(\tau_p^\#)}(k_i,R_{ki})=KR_{\tau_p^\#}(k_i,R_{ki})$若 $k_i\leqslant k$;

(2) $KR_{\hat{RefL}_1(\tau_p^\#)}(k+1,R_{k+1})=\varnothing$;

(3) $\bigcup KR_{\hat{RefL}_1(\tau_p^\#)}(k+2,R_{k+1}^*)=KR_{\tau_p^\#}(k+1,R_k^*)$。

内存泄露是指内存中单元没有被指针变量或其他内存结点引用，成为孤立结点。由于堆操作程序的操作内存结点具有局部性，在内存抽象模型中，指针遍历距离指针值为 $k+1$ 及以后结点时，首先遍历距离 $k+1$ 的内存结点。精化模型时，为了不产生漏报，将所有距离指针中值为 $k+1$ 的指针别名集设置为空。因此，假设精化前摘要结点的指针别名集中元素个数为 m，则简化的面向内存泄露的内存模型精化操作可以产生 n^m 种情况。

定义 9(简化的非法指针解引用模型精化$\hat{\text{RefD}}$) k-limit 框架下指针 p 的扩展结构表示为：$\tau_p^\# =\{0:\langle R_0;2^{PVar}\rangle;1:\langle R_1;2^{PVar}\rangle;\cdots;k_i:\langle R_{ki};2^{Pvar}\rangle;\cdots;k:\langle R_k;2^{Pvar}\rangle;k+1:\langle R_k^*;2^{PVar}\rangle;\}$，模型精化后，在 $k+1$ 框架下扩展结构$\hat{RefD}_1(\tau_p^\#)$，假设聚集结构类型的指针成员为：$f_1,f_2,\cdots,f_n$，转换过程如下：

(1) $KR_{\hat{RefD}_1(\tau_p^\#)}(k_i,R_{ki})=KR_{\tau_p^\#}(k_i,R_{ki})$，若 $k_i\leqslant k$;

(2) $\bigcup KR_{\hat{RefD}_1(\tau_p^\#)}(k+1,R_{k+1})=KR_{\tau_p^\#}(k+1,R_k^*)$;

(3) $KR_{\hat{RefD}_1(\tau_p^\#)}(k+2,R_{k+1}^*)=\varnothing$。

非法指针解引用是指被解引用的指针所指向的内存结点不存在。非法指针解引用往往是由于释放某个内存单元，其他指向该内存单元的指针仍然保存该内存地址。精化模型时，为了不产生漏报，尽可能地保留被修改内存结点的指针别名集，所有距离指针值为 $k+1$ 的指针别名集保留为相对应的摘要结点上别名集。因此，假设精化前摘要结点的指针别名集中元素个数为 m，则简化的非法指针解引用模型精化操作可以产生 n^m 种情况。

总之，面向特定内存安全性的简化模型精化操作能够将结果由 $(2n)^m$ 种情况降低为 n^m 种情况。

3 内存安全性检测算法

3.1 堆操作程序基本语义

基于 k-limit 内存框架下堆内存抽象状态的指针赋值语句的操作语义如图 7 所示。符号 $S^\#$ 表示每个指针变量与其对应扩展结构 $\tau_p^\#$ 的映射环境。公式中，k 的范围为：$0,1,2,\cdots,k,k+1$；R_k 表示所有路由次数为 k 的解引用路径，符号 $R_k+\rightarrow f_i$ 表示 R_{k+1} 中那些最后一次解引用的指针域为 f_i 的解引用路由路径；符号 $p\rightarrow f_i+R_k$ 表示 R_{k+1} 中那些第一次解引用的指针域为 f_i 的解引用路由路径；$R_k\uparrow y\rightarrow f_i$ 表示 R_k 中那些以 $y\rightarrow f_i$ 开头的解引用路由路径。

$$
\begin{aligned}
&(1)[\![p=null]\!](S^\#)=\{S^\#\left[KR_{\tau_p}(k,R_k)\leftarrow KR_{\tau_p}(k,R_k)-\{p\}\mid\tau_x^\#\in S^\#-\{\tau_p^\#\},KR_{\tau_p}(k,R_k)\leftarrow\perp\right]\}\\
&(2)[\![p\rightarrow f_i=null]\!](S^\#)=\{S^\#\left[KR_{\tau_x}(k+1,R_k+\rightarrow f_i)\leftarrow\perp\mid x\in\{x\mid p\in KR_{\tau_x}(k,R_k)\},KR_{\tau_y}(k,R_k\uparrow y\rightarrow f_i)\leftarrow\perp\mid y\in\{p\}\cup KR_{\tau_x}(0,p)\right]\}\\
&(3)[\![p=q]\!](S^\#)=\{S_1^\#\left[KR_{\tau_p}(k,R_k)\leftarrow KR_{\tau_q}(k,R_k)\right]S_1^\#\in[\![p=null]\!]S^\#\}\\
&(4)[\![p=q\rightarrow f_i]\!](S^\#)=\{S_1^\#\left[KR_{\tau_p}(0,p)\leftarrow KR_{\tau_q}(1,q\rightarrow f_i),KR_{\tau_q}(1,q\rightarrow f_i)\leftarrow KR_{\tau_q}(1,q\rightarrow f_i)\cup\{p\}\right]\mid S_1^\#\in[\![p=null]\!]S^\#\}\\
&(5)[\![p\rightarrow f_i=q]\!](S^\#)=\{S_1^\#\left[KR_{\tau_p}(1,p\rightarrow f_i)\leftarrow\{q\}\cup KR_{\tau_q}(0,q_i),KR_{\tau_q}(k+1,p\rightarrow f_i+R_k)\leftarrow KR_{\tau_q}(k,R_k)\right]\mid S_1^\#\in[\![p\rightarrow f_i=null]\!]S^\#\}\\
&(6)[\![p=malloc]\!](S^\#)=\{S_1^\#\left[KR_{\tau_p}(0,p)\leftarrow\varnothing,KR_{\tau_p}(1,R_1)\leftarrow\perp\right]S_1^\#\in[\![p=null]\!]S^\#\}\\
&(7)[\![p=free()]\!](S^\#)=\{S^\#\left[KR_{\tau_x}(k,R_k)\leftarrow\Theta\mid p\in KR_{\tau_x}(k,R_k),KR_{\tau_y}(0,y)\leftarrow\Theta\mid y\in KR_{\tau_p}(0,p),KR_{\tau_p}(0,p)\leftarrow\perp\right]\}
\end{aligned}
$$

图 7 指针赋值语句操作语义

复合语句的基本语义如图 8 所示。顺序语句的执行过程是：在抽象状态 $S_0^\#$ 下先执行语句 $Stmnt_1$ 得到中间状态 $S_1^\#$，然后在中间状态 $S_1^\#$ 下执行语句 $Stmnt_2$；对于 switch 语句，首先在当前堆内存抽象状态 $S_0^\#$ 下求解 switch 语句的条件，然后根据条件值选择执行下一条语句。

$$
\frac{S_1^\#=[\![A\text{sgn}\,Stmnt_1]\!](S_0^\#)\quad S_2^\#=[\![A\text{sgn}\,Stmnt_2]\!](S_1^\#)}{S_2^\#=[\![A\text{sgn}\,Stmnt_1;A\text{sgn}\,Stmnt_2]\!](S_0^\#)}
$$

$$
\frac{[\![e]\!](S_0^\#)=[\![c_k]\!](S_0^\#)}{[\![switch\ \ e\ \ (c_1:n_1,\ldots,c_k:n_k,\ldots)]\!](S_0^\#)\mapsto[\![n_k]\!](S_0^\#)}
$$

图 8 复合语句操作语义

过程间语句(包括过程调用和返回语句)的操作语义如图 9 所示。被调用过程中，$p_1,p_2,\cdots,p_k$ 为形式参数，ret_f 为返回值；调用过程中，$LVar_f$ 为局部指针变量集，$GVar_f$ 分全局指针变量集。被调用过程入口处的堆内存抽象状态为 $S_{0f}^\#$，$S_{1f}^\#$表示被调用过程在状态 $S_{0f}^\#$下返回的抽象状态。$e=f(e_1,e_2,\cdots,e_k)$为函数调用语句，该处的堆内存抽象状态为 $S^\#$，$S_f^\#$为调用函数中执行函数调用语句后的状态。

过程调用语句的状态迁移过程如下：首先，将全局指针变量的扩展结构传递给被调用过程，将实际参数的扩展结构传递给形式参数的扩展结构，并作为被调用入口处的初始堆内存状态，将被调用过程的局部指针变量初始化为空；然后，遵循堆操作程序过程内分析算法得到堆内存状态 $S_{1f}^\#$；最后分别将指针类型的返回值和全局指针变量的扩展结构传递给调用过程。调用过程中，其他局部指针变量的扩展结构在过程调用前后保持不变。函数返回语句 return e 将指针变量 e 的

指针扩展结构赋值给函数的返回值，全局变量返回给调用过程，被调用过程的局部指针变量重新赋值为空。

$$\begin{cases} [\![g_k]\!](S^*_{0f}) = [\![g_k]\!](S^*) & \text{if } g_k \in GVar \\ [\![p_k]\!](S^*_{0f}) = [\![e_k]\!](S^*) & \\ [\![l_k]\!](S^*_{0f}) = \bot & \text{if } l_k \in LVar \\ S^*_{1f} = [\![f(p_1, p_2, \ldots, p_k)]\!](S^*_{0f}) & \\ [\![g_k]\!](S^*_1) = [\![g_k]\!](S^*_{1f}) & \text{if } g_k \in GVar \\ [\![x_k]\!](S^*_1) = [\![x_k]\!](S^*) & \text{otherwise} \\ [\![e]\!](S^*_1) = [\![ret_f]\!](S^*_{1f}) & \\ S^*_1 = [\![e = f(e_1, e_2, \ldots, e_k)]\!](S^*) & \end{cases} \qquad \begin{cases} [\![ret_f]\!](S^*_{1f}) = [\![return\ e]\!](S^*_{1f}) & \\ [\![g_k]\!](S^*_1) = [\![g_k]\!](S^*_{1f}) & \text{if } g_k \in GVar \\ [\![l_k = null]\!](S^*_{1f}) & \text{if } l_k \in LVar_f \end{cases}$$

图 9　过程调用语句操作语义

3.2　内存安全性

堆操作程序的内存安全性主要包括：内存泄露、空指针引用、悬挂指针引用、多次释放。程序中只有指针赋值语句才能改变内存的形态和指针间的指向关系，这可能导致上述内存错误。

函数 $hleakCheck(s, S^\#)$ 的功能是：检测堆操作程序中指针赋值语句 s 在抽象状态 $S^\#$ 下是否发生内存泄露错误，根据语句 s 的类型可以分为：

(1) p=null，$p=q$，$p=q\rightarrow f_i$，p=malloc

这 4 条语句修改了指针 p 的指向关系，$hleakCheck(s, S^\#)$=true，若 $KR^\#_{\tau_p}(0, p)=\varnothing$；

(2) $p\rightarrow f_i$=null，$p\rightarrow f_i=q$

这 2 条语句修改了指针 p 指向聚集内存单元中指针域 f_i 的指向关系，$hleakCheck(s, S^\#)$=true，若 $KR^\#_{\tau_p}(1, p\rightarrow f_i)=\varnothing$；

(3) p=free()

$hleakCheck(s, S^\#)$=true，若 $\exists r_1 \in R_1.\ KR^\#_{\tau_p}(1, r_1)=\varnothing$；

函数 $npdCheck(s, S^\#)$ 的功能是：检测堆操作程序指针赋值语句 s 在抽象状态 $S^\#$ 下是否发生空指针解引用错误，根据语句 s 的类型可以分为：

(1) $p\rightarrow f_i$=null

$npdCheck(s, S^\#)$=true，若 $KR^\#_{\tau_p}(0, p)=\bot$；

(2) $p=q\rightarrow f_i$

$npdCheck(s, S^\#)$=true，若 $KR^\#_{\tau_q}(0, q)=\bot$；

(3) $p\rightarrow f_i=q$

$npdCheck(s, S^\#)$=true，若 $KR^\#_{\tau_p}(0, p)=\bot$。

函数 $dpdCheck(s, S^\#)$ 的功能是：检测堆操作程序指针赋值语句 s 在抽象状态 $S^\#$ 下是否发生悬挂指针解引用错误，根据语句 s 的类型可以分为：

(1) $p\rightarrow f_i$=null

$dpdCheck(s, S^\#)$=true，若 $KR^\#_{\tau_p}(0, p)=\Theta$；

(2) $p=q\rightarrow f_i$

$dpdCheck(s, S^\#)$=true，若 $KR^\#_{\tau_q}(0, q)=\Theta$；

(3) $p\rightarrow f_i=q$

$dpdCheck(s, S^\#)$=true，若 $KR^\#_{\tau_p}(0, p)=\Theta$。

函数 $dfCheck(s, S^\#)$ 的功能是：检测堆操作程序指针赋值语句 s 在抽象状态 $S^\#$ 下是否发生多次释放错误。因此，只有内存释放语句才能引起多次释放错误，$dfCheck(p$=free$(), S^\#)$=true，若 $KR^\#_{\tau_p}(0, p)=\bot \vee KR^\#_{\tau_p}(0, p)=\Theta$。这里，$KR^\#_{\tau_p}(0, p)=\bot$ 表示指针 p 指向的内存单元已经被指针 p 释放；$KR^\#_{\tau_p}(0, p)=\Theta$ 表示指针 p 指向的内存单元已经被其他指针释放。

3.3　算法框架

本文实现了基于 k-limit 内存抽象框架下的数据流迭代算法，得到每个可达的程序点关于抽象状态的不动点，在迭代过程中自动检测内存的安全性，该算法框架如图 10 所示。

```
input: the whole program HP, optional local memory abstraction Pre ;
output: whether there are any memory-related errors in heap-
manipulating programs;
01: generate the function call graph FCG for HP;
02: generate the call flow graph CFG_f for each function f;
03: pick up a function f from top to down in FCG;
04: Abs(n) =⊥, ∀n ∈ CFG_f ;
05: W.push(⟨entryNode, Pre⟩) ;
06: while W is not empty;
07:   ⟨s, S*⟩ = W.pop() ;
08:   switch ( s )
09:     case AsgnStmt :
10:       if hleakCheck(s, S#) then
11:         hleakStack.push(s, S#);
12:       if npdCheck(s, S#) then
13:         npdStack.push(s, S#);
14:       if dpdCheck(s, S#) then
15:         dpdStack.push(s, S#),
16:       if dfCheck(s, S#) then
17:         dfStack.push(s, S#),
18:       S#1 = [[s]]S# ;
19:     break;
20:     case SwitchStmt, CallStmt, ReturnStmt:
21:       S#1 = [[s]]S#;
22:     break;
23:     default:
24:       S#1 = [[s]]S# ;
25:   for all s' ∈ Succ(s) do
26:     S#new = Saturate(Join(S#1, Abs(s'))) ;
27:     if S#new != Abs(s') then
28:     W.push⟨s, Abs(s')⟩;
29:   end for
30: end while
```

图 10　数据流迭代算法框架

迭代过程使用队列存储结果，主要操作包括：

$pop()$：从队列的首部弹出数据项；

$push()$：将数据项存入队列的首部。

该算法总共使用 5 个队列：W 存储待迭代计算的语句及其抽象状态；$hleakStack$ 存储可能发生内存泄露的语句及其状态；$npdStack$ 存储可能发生空指针引用的赋值语句及其状态；$dpdStack$ 存储可能发生悬挂指针引用的赋值语句及其状态；$dfStack$ 存储可能发生多次释放的语句及其状态。公式 $Abs(s)$ 表示语句 s 点的抽象状态；公式 $Succ(s)$ 表示语句 s 的所有后继语句集。

结束语　程序内存安全性静态分析的相关研究与工具主要包括：Xu Z. X. 等人[12]提出了基于约束求解器 CVC3 路径敏感的过程间内存泄漏检测算法；Y. Jung 和 K. Yi[13]在完全自动化的静态分析工具(SPARROW)上，设计了基于逃逸模型的参数化函数摘要技术；除此之外，SATURN 工具[6]将内存泄漏问题规约为布尔公式的可满足性问题，然后使用 SAT 求解器判断是否存在内存泄漏错误；Clouseau[14,15]是基于所有权转移(ownership transfer)描述释放堆内存的指针变量，用以构造所有权约束系统来检测内存泄漏错误。Dillig[16]等人基于不一致推理方法，提出了精确检测 C 程序空指针引用的形式化框架。

本文提出了一种面向堆操作程序内存安全性域敏感的 k-limit 内存抽象模型，并且设计了正规抽象和精化操作来支持该模型精度动态可调，最后提出了统一的内存安全性数据流迭代检测算法。进一步的工作包括：使用该内存模型分析

(下转第 151 页)

RPO的构建、证明及一致性条件下的同前相关推出的证明。另外，对其表达能力进行了相应的讨论。同纯 Bigraph 一样，嵌套赋类 Bigraph 是嵌套赋类位置图与连接图的组合。除了与纯 Bigraph 中相应概念的比较外，本文也给出了嵌套赋类 Bigraph 中一些重要的定义和命题。如何利用嵌套赋类 Bigraph 模型对分布移动式系统进行建模以检验模型的表达能力以及对连接图的赋类进行研究，都是值得进一步关注的问题。

参 考 文 献

[1] Milner R. The Space and Motion of Communicating Agents [M]. Cambridge: Cambridge University Press, 2009

[2] Jensen O H. Mobile processes in bigraphs [D]. Dept. of Computer Science, Aalborg University, 2006

[3] Jensen O H, Milner R. Bigraphs and mobile processes (revised) [R]. UCAM-CL-TR-580. University of Cambridge Computer Laboratory, Cambridge, 2004

[4] Milner R. Local Bigraphs and Confluence: Two Conjectures [J]. Amadio R, Phillips I, eds. Proceedings of the 13th International Workshop on Expressiveness in Concurrency, Electronic Notes in Theoretical Computer Science, 2006, 175(3): 65-73

[5] Grohmann D, Miculan M. Directed bigraphs[C]// Proceedings of 23rd MFPS Conference. Electronic Notes in Computer Science, 2007, 173: 121-137

[6] Grohmann D, Miculan M. Reactive systems over directed bigraphs [C]// Caires L, Vasconcelos V T, eds. Proceedings of the 18th International Conference on Concurrency Theory, volume 4703 of Lecture Notes in Computer Science. Springer-Verlag, 2007: 380-394

[7] Grohmann D, Miculan M. An algebra for directed bigraphs [J]. Mackie I, Plump D, eds. Proceedings of the 4th International Workshop on Computing with Terms and Graphs 2008, Electronic Notes in Theoretical Computer Science, 2008, 203(1): 49-63

[8] Grohmann D, Miculan M. Controlling resource access in directed bigraphs[C]// Ermel C, de Lara J, Heckel R, eds. Proceedings 7th International Workshop on Graph Transformation and Visual Modeling Techniques. Electronic Communications of the EASST, 2008, 10: 1-21

[9] Leifer J J, Milner R. Transition systems, link graphs and petri nets [R]. UCAM-CL-TR -598. Computer Laboratory, University of Cambridge, 2004

[10] Milner R. Pure bigraphs [R]. UCAM- CL-TR-614. Computer Laboratory, University of Cambridge, 2005

[11] Birkedal L, Debois S, Hildebrandt T. Sortings for reactive systems [C]// CONCUR06, Lecture Notes in Computer Science. Springer, 2006, 4137: 248-262

[12] Elsborg E. Bigraphs: Modeling, Simulation, and Type Systems [D]. Copenhagen, IT University of Copenhagen, 2009

[13] Shane O C. Kind bigraphs: Static theory [R]. Report 36. Trinity College Dublin, Computer Science Department, 2005

[14] Shane O C. Kind bigraphs [J]. Electronic Notes in Theoretical Computer Science, 2009: 361-377

(上接第 114 页)

其他程序相关的性质，比如代码克隆检测等。

参 考 文 献

[1] Chou A, Yang J, Chelf B, et al. An empirical study of operating systems errors[C]// Proceedings of the Eighteenth ACM Symposium on Operation Systems Priciples. New York, NY, USA: ACM, 2001: 73-88

[2] 梅宏，王千祥，等. 软件分析技术进展 [J]. 计算机学报，2009，32(9)：1697-1710

[3] Hasting R, Joyce B. Purify: Fast detection of memory leaks and access errors[C]// Proceedings of the Winter USENIX Conference. San Francisco, USA: Winter, 1992: 125-136

[4] Havelund K, Rosu G. Monitoring Java programs with Java PathExplorer[C]// Proceedings of the 1st Workshop on Runtime Verification. Paris, France, 2001

[5] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation [C]// Proceedings of the 2007 Programming Language Design and Implementation Conference. New York, NY, USA: ACM, 2001: 89-100

[6] Evans D. Static detection of dynamic memory errors[C]// SIGPLAN Conference on Programming Language Design and Implementation. Philadelphia, USA: ACM, 1996: 44-53

[7] Xie Y, Aiken A. Saturn: A scalable framework for error detection using Boolean satisfiability [J]. ACM Trans. Program. Lang. Syst., 2007, 29(3)

[8] Nagarakatte S, Zhao J, Martin M M, et al. SoftBound: highly compatible and complete spatial memory safety for c [J]. SIGPLAN Not., 2009(44): 245-258

[9] Cooper K D, Harvey T J, Kennedy K. Iterative data-flow analysis, revisited [R]. TR 04-100. Rice Technical Report

[10] 董龙明，王戟，陈立前，等. 基于局部堆内存抽象表示的堆操作程序内存泄露检测[J]. 计算机研究与发展

[11] Dong L M, Dong W, Chen L Q. Invalid Pointer Dereferences Detection for CPS software based on Extended Pointer Structures [C]// The 2nd International Workshop on Safety and Security in Cyber-Physical Systems. Washington, D. C., USA: IEEE, 2012

[12] Xu Z X, Zhang J. Path and context sensitive inter-procedural memory leak detection[C]// Proceedings of the 2008 The Eighth International Conference on Quality Software. Washington, DC, USA: IEEE Computer Society, 2008: 412-420

[13] Jung Y, Yi K. Practical memory leak detector based on parameterized procedural summaries[C]// Proceedings of the 7th International Symposium on Memory Management. New York, NY, USA: ACM, 2008: 131-140

[14] Heine D L, Lam M S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector[C]// Proceedings of the ACM Conference on Program Language Design and Implementation. New York, NY, USA: ACM, 2003: 168-181

[15] Heine D L, Lam M S. Static detection of leaks in polymorphic containers[C]// Proceeding of the International Conference on Software Engineering. New York, NY, USA: ACM, 2006: 252-261

[16] Dillig I, Dillig T, Aiken A. Static error detection using semantic inconsistency inference [J]. SIGPLAN Not., 2007, 42: 435-445