计算机科学
COMPUTER SCIENCE

过程间流敏感的指针分析技术研究

帅东昕，葛丽丽，谢金言，张迎周，薛渝川，杨嘉毅，密杰，卢跃

引用本文

帅东昕，葛丽丽，谢金言，张迎周，薛渝川，杨嘉毅，密杰，卢跃. 过程间流敏感的指针分析技术研究[J]. 计算机科学，2023，50(12)：1-13.

SHUAI Dongxin, GE Lili, XIE Jinyan, ZHANG Yingzhou, XUE Yuchuan, YANG Jiayi, MI Jie, LU Yue. Survey of Interprocedural Flow-sensitive Pointer Analysis Technology[J]. Computer Science, 2023, 50(12): 1-13.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.221000195

过程间流敏感的指针分析技术研究

帅东昕　葛丽丽　谢金言　张迎周　薛渝川　杨嘉毅　密　杰　卢　跃
南京邮电大学计算机学院　南京 210023
(1184822826@qq.com)

摘　要　指针分析技术是一种基础的静态程序分析技术,也是软件安全方向的研究热点之一,在软件缺陷检测、恶意代码分析、程序验证、编译器优化等应用场景中发挥着重要的作用,指针分析的精度在这些应用场景中至关重要。流敏感分析和过程间分析是提高指针分析精度最有效的两种技术。文中对现有的提高过程间流敏感指针分析精度的技术进行总结,从为提高精度所消除的信息入手,将分析方法分为两大类:一类是消除分析中的虚假信息,以避免指向信息沿虚假的返回路径或是虚假调用关系传播;另一类是消除分析中保守的指向关系,在每个程序点处根据设置的规则尽可能确定指针的唯一指向,而不是笼统地计算指针的多个可能指向。据此,详细比较了过程间流敏感指针分析技术的异同,并对指针分析技术未来的研究方向进行了展望。

关键词:指针分析;流敏感分析;过程间分析;精度优化;调用上下文
中图法分类号　TP311

Survey of Interprocedural Flow-sensitive Pointer Analysis Technology

SHUAI Dongxin,GE Lili,XIE Jinyan,ZHANG Yingzhou,XUE Yuchuan,YANG Jiayi,MI Jie and LU Yue
School of Computer Science,Nanjing University of Posts and Telecommunications,Nanjing 210023,China

Abstract　Pointer analysis technology is a basic static program analysis technology,it has always been one of the research hotspots in the direction of software security,which plays an important role in software defect detection,malware analysis,program verification,compiler optimization and other application scenarios. The accuracy of pointer analysis in these application scenarios is crucial. Flow-sensitive analysis and interprocedural analysis are the two most effective techniques for improving the accuracy of pointer analysis. This paper summarizes the existing techniques for improving the accuracy of interprocedural flow-sensitive pointer analysis,starting from the information eliminated by methods to improve accuracy,and it is divided into two categories. One is to eliminate false information in the analysis to avoid the propagation of pointing information along a false return path or false call relations. The other is to eliminate the conservative points-to relations,so that to determine the unique location assigned to the pointer at each program point,rather than generally calculating the possible multiple points of the pointer. Accordingly,this paper compares the similarities and differences of the interprocedural flow sensitive pointer analysis technology in detail,and outlines the future research direction of the pointer analysis technology.

Keywords　Pointer analysis,Flow-sensitive analysis,Interprocedural analysis,Precision optimization,Calling context

1　引言

指针可以有效地表示复杂的数据结构,可用于方法(过程)的参数传递,并帮助人们更加灵活地使用方法。在程序中使用指针,可以使程序设计更加灵活、实用、高效。在软件分析中,指针信息是获取程序的控制流、数据流等信息的基础,因此指针分析是许多软件分析工具不可缺少的部分,例如构建程序的函数调用图(Call Graph,CG),就需要指针信息才能判断函数的调用关系。除此之外,指针分析的结果可以指导软件缺陷检测、恶意代码分析、程序验证、编译器优化等工具的执行,而指针分析的精度也会进一步影响这些工具的效率和准确性,低精度的指针分析会给这些分析工具带来高误报率和漏报率。因此,精确、高效的指针分析可以保证软件的可靠性。本文主要讨论如何提高指针分析的精度,我们发现,过程间分析和流敏感分析是提高指针分析精度的有效方法。

过程内分析只考虑一个过程/函数内部语句的数据流信息,而过程间分析则需要考虑数据流在整个程序中的传播。指针分析是一类特殊的数据流问题,指针信息既可以从调用者传播到被调用者,又可以从被调用者传播到调用者,选取的过程间分析策略不恰当会对指针分析的精度产生影响,指针分析精度降低反过来也会影响到过程间分析的精确性。

流不敏感的分析会将一个过程内所有相关的语句汇总分析得到一个结果,而流敏感的分析则会考虑程序的控制流,按照程序语句的执行顺序和分支结构为每个程序点都生成一份

到稿日期:2022-10-24　返修日期:2023-03-17
通信作者:张迎周(zhangyz@njupt.edu.cn)

结果，因此流敏感的指针分析比流不敏感的分析精度更高。

指针分析在程序分析、程序安全方面有着广泛应用，国内外学者已经从不同角度对指针分析的精度、优化技术等进行了分析总结。Hind 等[1]通过实验对 Address-taken，Steensgaard[2]，Andersen[3]，Burke[4]，Choi[5]这 5 种当时主流的指针分析算法进行了对比，通过 Mod/Ref 分析、存活变量分析、到达定义分析、常量传播分析这 4 种客户端分析方法对 5 种算法的精度进行评估。Hind 等[6]讨论了指针分析领域普遍关注的问题和未解决的开放性问题，Chen[7]讨论了基于包含的指针分析的优化技术，根据优化的阶段将其分为在线优化（约束求解过程中）和离线优化（约束求解之前）。在线优化阶段有三大类方法，分别是约束图上强连通分量的检测与消除、消除冗余的指向集传递以及工作集节点的选取顺序；离线优化主要是根据指针等价和位置等价进行变量的替换，文中还对所提及的几种优化算法进行了实验对比。Yao[8]讨论了面向软件可信性的指针分析的关键技术。Li 等[9]主要从上下文的表示方法、指针分析的实现方法、指针分析的优化方法、别名分析、指针分析的评估指标方面进行了分析总结，用很大篇幅讨论了在面向对象程序中上下文元素（调用点、对象、类型、this 以及这几种元素的结合）的选取对面向对象程序指针分析精度和效率的影响。

上述文章对指针分析进行综述，但仍存在不足之处，且目前尚未有文章专门对过程间流敏感指针分析的精度优化问题进行总结与梳理。Hind 等[1,6]进行综述的时间较早，近年来指针分析工作的进展较快，难以概括最新进展。Chen 等[7]仅讨论了流不敏感指针分析中的一类：基于包含的指针分析。Yao 等[8]对面向软件可信性的流敏感、流不敏感、上下文敏感的指针分析技术进行了分类介绍，但是没有对新方法进行总结。Li 等[9]对面向对象的上下文敏感指针分析进行了多方面的总结分析，但是对包含流敏感信息的指针分析没有过多提及。

通过静态分析获得完整精确的指向信息已经被证明是不可判定的[10-12]，因此许多指针分析算法都在精度和效率之间有不同的取舍。本文重点关注提高过程间流敏感指针分析精度的措施及角度，并根据这些角度对提高精度的方法进行总结。

本文第 2 章介绍指针分析的相关概念、影响指针分析精度的因素，以及过程间流敏感指针分析提高精度的措施；第 3 章介绍消除虚假信息相关的算法；第 4 章介绍消除保守指针信息相关的算法；第 5 章介绍指针分析领域其他影响精度的问题；最后总结全文并阐述过程间流敏感指针分析领域潜在的研究方向。

2 背景介绍

2.1 指针分析相关概念

从广义的角度来看，指针分析指向分析、逃逸分析这 3 种分析[13]。别名分析（Alias analysis）[14]的目的是确定一对指针表达式是否互为别名，即两个指针是否指向同一个内存位置；指向分析是计算指针变量在运行时可能指向的一个或多个内存位置；逃逸分析[15-16]的目的是检测某块内存是否逃逸出特定的范围。在实际的程序分析工作中，不同类的分析通常不是单独存在的，各类指针分析之间紧密相关。这 3 种分析能从不同方面保障软件的可信性。如图 1(a)所示，别名分析可以得出在第 7 行指针 p 和 q 是一组别名，第 8 行 p 和 r 是一组别名；而指向分析则会得出第 6 行 $pts(p)=\{x\}$，$pts(q)=\{y\}$，$pts(r)=\{z\}$，第 7 行 $pts(p)=pts(q)=\{y\}$，第 8 行 $pts(p)=pts(r)=\{z\}$。

```
1  int main(){
2    int  *p,*q,*r;
3    int x,y,z;
4    p=&x;
5    q=&y;
6    r=&z;
7    p=q;
8    p=r;
9    return 0;
10 }
```

(a)C 语言示例程序

```
1  public class ObjectReturn{
2    public User createUser(){
3      User user=new User();
4      return user;
5    }
6  }
```

(b)Java 语言示例程序

图 1 含指针的程序示例

Fig. 1 Example of programs with pointers

如图 1(b)所示，对象 user 在方法 createUser 内部被创建，但是通过 return 语句传递到了方法外部，即对象 user 产生了逃逸。逃逸分析的目的是通过寻找逃逸对象来确定是否要将该对象分配到堆上。本示例中，如果将 user 分配到栈上，在 createUser 方法执行完毕后，user 将会出栈，那么方法外部就无法访问返回的 user 对象。

本文主要分析总结过程间流敏感的指向分析文章，也会涉及少量别名分析或逃逸分析的文章。

2.2 影响指针分析精度的因素

2.2.1 流敏感性

如果算法的分析过程不考虑控制流信息（包括语句的顺序和分支结构等），那么称算法为流不敏感算法，相反就称其为流敏感算法。流敏感的分析算法分析了每个或某些特定的程序点上相关变量的指向信息，而流不敏感的分析算法记录的是程序整体表现的指向信息，综合了所有执行语句的效果。流敏感算法的结果比流不敏感的算法结果更加精确，但是消耗空间较多，实现效率较低，实用性和可扩展性均不如流不敏感的算法。

流敏感算法主要有两种方法，一是基于数据流的迭代[17-18]，二是基于 SSA（Static Single Assignment，静态单赋值）[19]和定义引用链[20-22]。基于迭代数据流的方法需要分析程序控制流中每个节点“生成”和“注销”的指向信息，并构造数据流方程，在每个程序点处生成相应的指向分析结果。这种方法的优势在于精度非常高，但它会在每个程序点处都计算完整的指向信息，并随着控制流进行传递，因此会消耗大量的时间和空间。Whaley 等[18]采用集合包含的思想，在基于数据流的过程内指针分析的基础上，实现了流敏感的过程间

指针分析。基于 SSA 的方法首先将待分析程序转换为 SSA 形式，每次变量赋值后程序都会被赋予一个新变量名，然后再对转换后的程序进行流不敏感的指针分析。这种方式可以直接利用流不敏感指针分析算法，因此更简单也更受欢迎。Hardekopf 等[20-21]率先将稀疏分析引入指针分析领域，而后对该算法进行了改进，用不太精确的指针分析的定义-使用链来辅助实现流敏感的稀疏指针分析。这种方法极大地提高了指针分析的效率，但它并没有考虑上下文，因此在精度上有所欠缺。Sui 等[22]则为每种指令的指针信息都建立了公式，根据定义-使用链自底向上地进行按需分析。流不敏感的指针分析相比流敏感的指针分析在精度上有所欠缺，但分析速度快，在开源或者产品级高级编译器中被普遍使用[7]，它主要包括两类方法：基于包含的指针分析[3]和基于合并的指针分析[2]。其中基于包含的指针分析用集合包含的约束关系来表示指向关系，相比基于合并的指针分析来说精度更高，因此应用更加广泛。

Landi 等[23]的研究表明，如果存在动态存储，那么流敏感的指向分析算法是不可判定的。另一方面，Horwitz[10]证明，即使不存在动态存储，精确的流不敏感指向分析仍然是 NP 问题。

Samuel[24]提出了一种流敏感的基于约束的指向分析算法，实验结果表明，在所测试的 500 个程序上，流敏感算法的分析时间几乎是流不敏感的算法的 10 倍以上，但它能有效地提高精度。然而，由于缺乏针对精度提高程度的有效衡量标准与手段，流敏感算法在大规模程序分析精度方面的提升表现仍然没有定论。

2.2.2 上下文敏感性

上下文敏感的分析技术是提高过程间分析精度最有效的一种技术。过程间指针分析指考虑指向信息在多个过程间的传播，分析过程中会涉及指针在不同上下文中的指向差异。为了保证过程间指针分析的精度，在进行过程间指针分析时需要选择合适的上下文敏感分析技术。

上下文敏感分析技术用于区分不同的上下文信息。指向分析算法通过应用上下文敏感的分析技术来区分不同上下文中的全局指向信息。这些全局指向信息将会对过程内的各个局部变量的指向关系产生影响，如实参、调用接收者以及全局变量等的指向关系都会影响局部变量指向关系的计算结果，导致同一种方法在不同的调用上下文下可能有不同的表现。而在上下文不敏感的分析算法中，同一个过程在不同的调用上下文下数据流信息是相同的，因此其行为也是相同的。上下文不敏感算法对所有的调用进行统一处理分析，降低了分析的复杂度，提高了分析的效率。而上下文敏感算法对每一个上下文下对应的调用实例进行单独分析，从而提高了分析的精确度，但是在效率和复杂度方面也面临着的挑战。

上下文不敏感的指向分析的主要优势在于其具有较高的分析效率和可扩展性，有些工作通过类层次结构来提高指向分析的精度，但本质上这些工作仍然是上下文不敏感的。

随着程序规模进一步扩大，分析中需要记录的上下文数量也呈指数增长，部分学者提出了选择性上下文敏感(Selective Context-sensitive)的方法。在分析中，有些过程/函数中的指向关系并不受上下文的影响，因此可以通过机器学习[25]、模式匹配[26]、CFL 可达性[27-28]等方法将这些过程/函数筛选出来，然后对受调用上下文影响的过程/函数执行上下文敏感分析。Li 等[26]通过观察函数中的数据流，归纳出了 3 种模式，对象流过路径上经过的函数都需要进行上下文敏感的分析。Lu 等[27-28]提出了 EAGLE，在轻量级预分析期间，在进程中的变量/对象级别推理上下文无关语言(CFL)可达性，从而选定某些变量/分配点进行上下文敏感分析。

随着面向对象程序设计语言的流行，学者们也提出了许多上下文敏感性的变体。对象敏感性就是其中一种，对象指接收对象的分配点(创建对象的程序点)[29]。对于面向对象程序来说，对象敏感性比调用点敏感性更加精确、有效，且被认为是为 Java 提供良好精度的最有效的上下文敏感变体[30-35]。此外，上下文敏感性还有其他的变体，如类型敏感性[36]、this 敏感性[37]、参数敏感性[38]等。

2.2.3 域敏感性

域敏感分析可以区分结构对象的域。对于 C/C++语言，域敏感性的挑战在于如何获取某个域的地址，并将其存储到某个指针，然后通过 load 语句读取。Pierce 等[39]提出了 Pkh，采用基于域索引的抽象建模，使用唯一索引来区分对象的字段。Andersen 的约束图可以通过添加一个新的域约束来对域的地址指令建模，以便在约束解析期间导出。Mine[40]提出了一种对域和数组敏感的分析，该分析将域对象和数组访问转换为抽象解释框架中的指针算术。LPA[41]是一种面向循环的指针分析，用于自动 SIMD 矢量化。DSA[42]支持使用字节偏移对象建模的域敏感性，但是，该方法适用于 Steensgaard 指针分析，比 Andersen 指针分析精度稍低。

2.2.4 其他维度

除了上面提及的流敏感性、上下文敏感性、域敏感性以外，还有一些维度可以影响指针分析的精度，如路径敏感性[43]、状态敏感性[44]等。

2.3 过程间流敏感指针分析精度优化

本文并不是从上述角度对所讨论的过程间流敏感指针分析方法进行分类，而是从为提高精度所消除的信息入手，将分析方法分为两大类：一类是消除分析中的虚假信息，以避免指向信息沿虚假的返回路径或是虚假调用关系传播；另一类是消除分析中保守的指向关系，在每个程序点处根据设置的规则尽可能确定指针的唯一指向，而不是笼统地计算指针可能的多个指向，这样不仅可以逐一提高每个程序点处的指向分析精度，而且由于数据流传播的事实减少，还可以提高指针分析的效率。

虚假返回路径指在过程间分析中，由于存在一个函数被多次/多个函数调用的情况，因此指向信息可能会沿错误的返回路径将被调用函数的指向信息传递给其他调用函数。这是由于静态分析不随程序的执行进行分析，而且算法没有对不同的调用点做细致区分，导致指针流信息沿虚假返回路径传给了不存在的调用者，从而影响了分析结果的准确程度，这种问题也被称为调用上下文问题。随着程序规模的扩大，如果

在分析中未处理好虚假返回路径的问题，那么错误的指针信息会随着数据流一直向下传播，占用很多不必要的时间和空间。虚假返回路径的消除方法将在 3.1 节进行讨论。

虚假调用关系指生成的调用图中产生的许多不应存在的调用关系，而大部分虚假调用关系都是没有正确分析函数指针导致的，因此本文列举的文献主要是对函数指针分析进行的优化。函数指针是指向函数的指针变量。C 语言在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址，有了指向函数的指针变量后，就可以用该指针变量间接调用函数。进行过程间分析前通常需要先构建函数调用图，存在函数指针的情况下，调用图不能仅通过程序语义来构建，因为函数指针的调用点不能在编译时绑定到唯一的函数。如果保守计算函数指针的指向（假设函数指针指向程序中所有函数的集合，或是已获取地址的所有函数的集合），那么就会产生很多虚假的调用关系，导致调用图不精确，从而导致指向信息被传递给错误的函数[45-47]。即使存在一个虚假的调用关系，也会大大降低所获取的流信息和指向信息的质量；同时，由于每个函数都必须在调用上下文中进行分析，保守的函数指针信息可能会造成很大的时间和空间的浪费。虚假调用关系的消除方法将在 3.2 节中讨论。

保守的指向关系指在指针分析中不对每个程序点处的指向关系进行单独分析，只逐一地将新值添加到指向集中，不判断是否能够覆盖指向集中原先的内容，如流不敏感分析中就会存在很多保守的指向关系。而流敏感的指针分析中，会按照控制流的顺序对每条语句进行处理，分析每条语句生成和消除的指向关系，从而得知某个程序点处指针实际指向哪一个内存位置，然后根据语句内容用新的指向关系完全覆盖之前的指向关系，这在指针分析中被称为强更新。保守指向关系的消除方法将在第 4 章中讨论，但其中讨论的指针是普通指针，不包括函数指针，因为保守的函数指针信息相比普通的指针信息来说，不仅会影响过程内和过程间指向图的精度，还会导致调用图构建不精确，以至于指向信息沿虚假调用路径传播，保守的函数指针信息所引起的问题将在第 3.2 节中单独讨论。

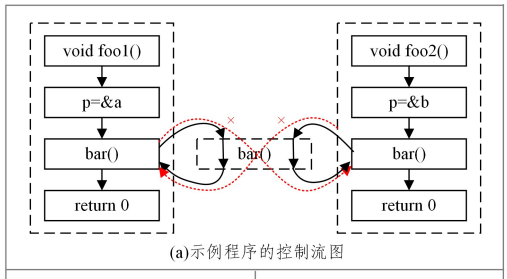
3 虚假信息的消除

在指针分析中消除虚假的信息可以避免指针指向在过程间沿虚假返回路径或虚假调用关系传播，提高过程间指针分析的精度，从而增加软件的可信性。消除虚假返回路径的关键在于如何将被调用函数的分析结果返回给正确的调用函数。解决这个问题的方法主要有两种，根据如何表示函数的调用环境（上下文）可以分为基于函数摘要的方法和基于调用串的方法。基于函数摘要的方法是以函数入口的分析结果来区分不同的上下文，而基于调用串的方法是使用调用栈中的某个序列作为上下文。此外还有基于克隆的方法和完全上下文敏感的方法。

3.1 虚假返回路径的消除

3.1.1 问题描述

如图 2 所示，函数 $foo1$ 和 $foo2$ 中分别产生了指向关系 $p\rightarrow a$ 和 $p\rightarrow b$，$p\rightarrow a$ 经调用路径 $foo1\rightarrow bar$、$p\rightarrow b$ 经调用路径 $foo2\rightarrow bar$ 传播到函数 bar 的入口处；函数 bar 不对指针 p 做任何修改，指针信息从 bar 出口处传播到 $foo1$ 和 $foo2$。



(a)示例程序的控制流图

```
1  void foo1(){
2    int p=&a;
3    bar();
4    return 0;
5  }
6  void foo2(){
7    int* p=&b;
8    bar();
9    return 0;
10 }
11 void bar(){
12   return 0;
13 }
```

(b)示例程序

行号	指向关系
4	pts(p)={a,b}
9	pts(p)={a,b}
12	pts(p)={a,b}

(c)不区分调用上下文的结果

行号	指向关系
4	pts(p)={a}
9	pts(p)={b}
12	pt([foo1,bar],p)={a}, pt([foo2,bar],p)={b}

(d)区分调用上下文的结果

图 2 虚假返回路径问题示例

Fig. 2 Example of spurious return path problem

如果不对调用点做细致区分，那么指向关系 $p\rightarrow a$ 和 $p\rightarrow b$ 会在 bar 函数入口处被合并，合并后的指向信息 $pts(p)=\{a,b\}$ 会沿返回路径 13→9 和 13→4 分别传递给 $foo1$ 和 $foo2$。这个结果不精确的原因是，存在两条虚假路径 2→3→11→12→13→9（$foo1\rightarrow bar\rightarrow foo2$）和 7→8→11→12→13→4（$foo2\rightarrow bar\rightarrow foo1$），这也被称为调用上下文问题（Calling Context Problem）。

如果对调用点做区分，指向关系 $p\rightarrow a$ 会沿 2→3→11→12→13→4（$foo1\rightarrow bar\rightarrow foo1$）传递，$p\rightarrow b$ 会沿 7→8→11→12→13→9（$foo2\rightarrow bar\rightarrow foo2$）传递，因此在程序点 4 和 9 处会分别得到指向关系 $pts(p)=\{a\}$ 和 $pts(p)=\{b\}$。

3.1.2 基于函数摘要的方法

基于函数摘要的方法是按照一定顺序遍历函数调用图，对每个函数进行过程内分析，收集其输入和输出之间的指向关系并生成摘要，再次遇到该函数的调用时，就可以直接根据函数的入口信息，利用这些摘要得到输出的指向信息。

基于摘要的方式有两种分析顺序，自底向上（Bottom-up）和自顶向下（Top-down）。用自顶向下的顺序进行分析时，可以一边构建函数调用图，一边进行指针分析，但这种方法的问题在于，它可能会在不同的上下文中重复分析同一种方法；而用自底向上的顺序进行分析时，需要预先构造一个函数调用图，在函数调用图的基础上进行指针分析，在大多数情况下都不会重复分析同一个函数，只会重复分析函数调用图中的强连通分量。因此，许多基于函数摘要的方法都采用自底向上的分析顺序。这种方法的问题在于，构造函数调用图需要用到指针信息，例如程序中包含函数指针，构建函数调用图时需要先分析该函数指针可能指向的函数地址，根据

这些地址才能确定调用图。

Fhndrich 等[48]在调用图上迭代地进行自上而下和自下而上的遍历来生成并更新摘要,自下而上遍历时使用摘要的符号表示,自上而下遍历时使用显式表示,它提供了迭代不动点算法,但是可能需要指数级的时间空间,其中过程上下文是抽象状态,而不是调用串。

此外,基于摘要的方法需要重点解决的一个问题是如何对在被调用函数中使用而在调用函数中定义的指针进行建模。参数化名字空间[49-54]是常用的一种方法,参数化名字空间引入了函数的间接参数对上述指针进行建模,如指针 p 的指向集为 $pts(p)=\{a\}$,函数 foo 的形参为指针 p,函数体内部通过 $*p$ 对 p 进行解引用,此时 a 在函数 foo 内部被使用,但其并没有通过形参直接代入,因此这种方法引入了间接参数 p_1 来表示 $*p$。引入间接参数有助于缓存以前的分析结果,并且它们将递归过程的当前调用的参数和局部变量与先前调用的参数和局部变量分开,从而提高精度。在对函数进行过程内分析时,将调用函数中的指向信息映射到被调用函数的参数化名字空间中,通过过程内分析得到被调用函数的指向信息,然后将含间接参数的指向信息进行反映射,这样就完成了函数摘要的实例化。

这种方法区分调用上下文的方式是:函数在每一个激活(Invocation)处均会形成特定的指针别名模式,每个别名模式都有一个指向输出与其对应,在处理函数调用时,会检查其别名模式能否与已经存在的别名模式相匹配,如果匹配则直接复用之前的结果。

这类方法中,Emami 等[50]用一个三元组$(x,y,D/P)$来表示指向信息,x 和 y 表示抽象栈位置,D 表示肯定的指向关系,P 表示可能的指向关系。偏传递函数(Partial Transfer Function,PTF)[53]惰性地创建这些间接参数,Chen 等[54]则将符号概率作为概率函数的值分配给过程入口处的每个指向关系,别名模式的匹配过程则是使用传递函数将符号概率替换为概率函数的值来计算结果。这类方法的问题在于,每个别名模式的摘要都是在分析时动态创建的,因此很难扩展到大规模程序中。

参数化名字空间这一类方法大多是将指向信息作为摘要,而近年来的研究更多的是将指向图作为摘要。由于指向图的规模会随程序规模扩大而呈指数增大,因此学者们也对基本的指向图进行了不同角度的改进。

Whaley 等[18]提出了同步进行指向分析和逃逸分析的方法,为每个函数构建了一个指向逃逸图,该方法的主要改进是从提取和表示程序已分析和未分析区域交互的角度对算法进行设计。

Qian 等[55]的函数摘要是指向图的形式,它首先遍历调用图生成每个方法的摘要,然后在总指向图和摘要图的节点上执行拓扑排序和循环消除,以加快对总指向图的传递闭包计算。

除了对分析顺序和摘要形式进行改进以外,学者们还针对实际分析中存在的问题从不同角度进行了改进。

David 等[56]针对动态指针分析提出了 Past-Sensitive Pointer Analysis(PSPA),将“过去”的分配点和“将来”的分配点区分开,以提升指针分析的精度。该算法在分析过程中不使用函数摘要,而只在顶级调用(Top-level Calls)之前使用。

Philipp 等[57]提出了 ModAlyzer,基于依赖关系的集成策略来生成类型层次结构、指向和调用图信息的摘要,而无须对丢失的代码进行假设,之后再使用这些预先计算的摘要可以有效地将大部分计算工作转移到离线阶段,这种方法相比 WPA 能节约 72%的时间,但不足之处在于无法有效分析程序中声明的常量指针和解决大量使用回调函数的问题。

Wang 等[58]同样也使用函数摘要来获取上下文信息,为获取流敏感指针信息,他们在控制流自动机上运行程序;同时,为提高效率,在分析过程中收集路径条件以过滤无法到达的路径,提出了一种多入口机制。他们的工具 TsmartGP 比 cppcheck 和 Clang 静态分析器更准确,能发现更多的指针相关漏洞。

函数摘要在程序分析领域应用广泛,主要用于提高过程间分析的效率,在遇到函数调用时,对被调用函数进行分析,然后将分析内容保存为函数摘要,在后续分析过程中,如遇到该函数的调用,都可以用函数摘要来代替对函数的分析,以避免重复分析。虽然函数摘要已经在过程间分析领域广泛使用,但目前还没有通用的方法来创建或者表示函数摘要,上述文献中的函数摘要也有多种形式,如指向图、记忆图等。

3.1.3 基于调用串的方法

基于调用串的方法通常以调用栈对应的某个序列作为上下文,比如调用栈中最近 k 个过程的序列或调用图上到某个过程的最近无环路径,然后将过程在不同上下文下的实例当作不同实体来处理。具体的操作是给函数/变量/指向关系添加调用路径信息,因此也被称为基于标签的方法[9]。在面向对象程序的指针分析中,由于对象的存在,标签可以有不同的类型,如对象、对象类型、this 等。但在过程间分析中,标签的类型通常只有调用点,因此本文将该类方法称为基于调用串的方法[59-62]。

如图 2(d)所示,从 $foo1$ 向 bar 传递的指向信息 $p\rightarrow a$ 可以表示为 $pt([foo1,bar],p)=\{a\}$,从 $foo2$ 向 bar 传递的指向信息 $p\rightarrow b$ 可表示为 $pt([foo2,bar],p)=\{b\}$,这样指针信息在返回调用者时,将按照调用路径匹配相应的路径,从而保证过程间指针分析的精确性。根据路径信息,只有 $p\rightarrow a$ 能返回 $foo1$,同样地,只有 $p\rightarrow b$ 能返回 $foo2$。

附加调用路径信息的方法在一定程度上消除了虚假返回路径所引起的不精确性,但它还存在一定的局限性,无法满足实用的指针分析算法的要求。要得到精确的指针信息,附加调用路径信息的方法需要考虑在指向关系中添加全部的路径信息。当调用深度加深时,路径信息将产生指数级别的增长,这种问题的解决方法通常是限制路径长度,即 k-limiting[63]。早期的研究通常将 k 定为 1[64],随着人们对指针分析精度的要求提升,许多方法被提出,可以对 k 进行选择。文献[65-67]使用标准的调用串方法。

Jeong 等[68]提出了一种启发式的方法,根据每个函数(方法)的原子特征生成布尔表达式,进而为每个函数选择不同的上下文深度(只能选择 0,1,2),分别进行不同精度的上下文敏感分析。

Hua 等[69]利用指针分析进行 UAF 缺陷检测，根据 UAF 缺陷的特性做了上下文约减，具体操作是以程序语句 malloc，use 和 free 为起点向上查找，以这 3 个语句为一组向上遍历，从一组语句开始的程序点为起点，添加调用信息的标签，从而避免完全的上下文敏感分析，也可以避免 k-limiting 方法中 k 的大小影响指针分析的精度。

Shivers[60]在 Sharir 等[59]的基础上提出了 k-CFA，k 代表分析方法中需要记录的调用序列的长度。Palsberg 等[61]实现了 1-CFA 的指向分析，Agesen[70]使用可能到达方法的参数类型组合来标记不同的上下文，之后 Grove 等[71]成功将两种方法结合起来使用。

3.1.4 其他消除虚假返回路径的方法

除了基于函数摘要和基于调用串的方法以外，还有一些方法可以解决虚假返回路径的问题，如基于克隆的方法[71-72]。这种方法会生成一种方法的多个实例，以便在每个不同的调用上下文中调用不同的实例，从而避免信息流向错误的上下文。Whaley 等[73]对基于克隆的方法进行改进，所提方案并不克隆方法的代码本身，而是为每个克隆生成一个单独的答案，同时利用 BDD 来对不同上下文（别名模式）进行排序，从而减少时间和空间的消耗。

Thiessen 等[74]将传统的调用串标记为对上下文的转换，传统的上下文表示是这些转换的输入-输出映射对的显式枚举，因此该文提出了转换字符串，可以将任何非空的上下文集合映射到由所有上下文组成的集合。文中还提出了相应的推断规则，能够有效减少不同上下文中指向关系的数量。

IFDS 框架[75-76]是一种经典的过程间程序分析技术，可以将过程间有限分布子集问题转化为超图上的图可达问题，图可达问题需要判断图中哪些路径是可行的，即哪些 return 和 call 语句是匹配的。IFDS 中使用了基于 CFL 可达性[76]的技术来区分数据流在调用点的传播路径。CFL 的做法是先在控制流边上标注 label，然后定义一门上下文无关语言去识别路径，如果能被这门语言识别，那么就称这是一条可实现的路径，如果不能被这门语言识别，那么就称这是一条不可实现的路径。这种方法的基本做法是，对每一个调用点，在它的 call edge 标上$_i$，它的 return edge 标上$_i$，其他的边则标上 e，如果一条路径的括号能够完全匹配，则说明这条路径是可达的。Boomerang 等[77]利用 IFDS 框架实现了按需的指针分析，对于每个查询，它首先执行后向分析以收集指向信息，然后执行前向分析，以确定指向当前分配点的所有可达图（Access Graph）。

Sui 等[78-82]近年来提出了一系列方法，通过快速但不够精确的指向分析（预分析）构建一个近似的定义-使用图/稀疏值流图（SVFG），并在其上进行流敏感分析。此类方法不需要将指向集在控制流图上的所有程序点传播，只需要维护对象指向集的子集并在 SVFG 上迭代地进行值流构造和指针分析，从而为两者提供越来越高的精度。为了消除虚假的返回路径，该方法对 Mod-Ref 分析进行了改进。保守的 Mod-Ref 分析假设传递到调用点的非本地位置集合可以直接或间接地由被调用点读取和修改，而改进后的 Mod-Ref 分析会根据其定义的规则对被调用函数执行更精确的分析，过滤虚假的 Mod-Ref 信息，从而获取更精确的定义-使用关系，而后在 SVFG 图中，用调用点的 ID 标识一对过程间定义点的值流。

Sun 等[83]提出了 DynaSens，它使用切片来自动调整分析的上下文敏感性，该算法是需求驱动的指针分析。在指向分析中，堆承载的数据流由 load 和 store 语句组成，这些堆承载的依赖关系很难解决，该方法利用基于需求驱动的切片分析来解决这种依赖关系。给定一个指向查询，相关过程的集合由切片分析识别并由指向分析上下文敏感地处理。

3.2 虚假调用关系的消除

3.2.1 问题描述

如图 3(a)所示，第 5 行定义了一个全局函数指针 fp。在 *main* 函数体中，第 9 行 fp 指向函数 *func*1 的地址，第 11 行 fp 指向函数 *func*2 的地址，第 13 行对函数指针 fp 进行调用。对 *main* 函数进行初步指针分析后，可以得到 fp 可能指向 *func*1 或 *func*2，因此生成调用图时，*main* 函数内会分别调用 *func*1 和 *func*2。同样地，*func*1 函数内也调用了函数指针 *fp*，即调用图中 *func*1 函数内也会分别调用 *func*1 和 *func*2，如图 3(*b*)所示。

但实际上，通过 fp 调用 *func*1 函数时，fp 的指向为 *func*1 函数的地址，因此在 *func*1 函数体内对函数指针 fp 的调用只可能为 *func*1，不可能为 *func*2，实际的调用图应该如图 3(c)所示。

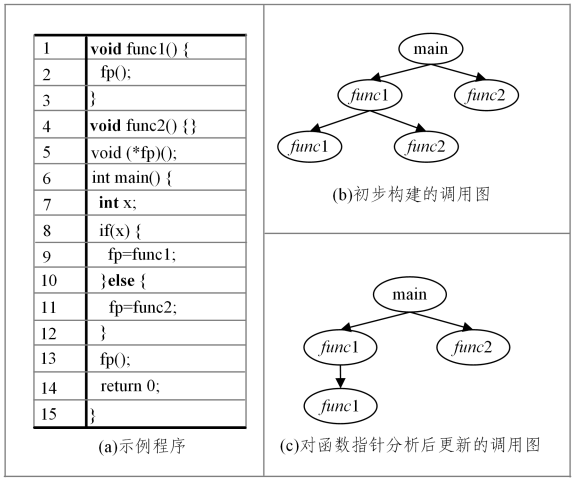


图 3 含函数指针的程序示例

Fig. 3 Example of program with function pointers

3.2.2 同时进行指针分析和构建调用图

同时进行指针分析和构建调用图（on-the-fly points-to analysis）是最常见的处理函数指针的方法。首先构建一个初始的（不完整的）函数调用图并进行指向分析，在遇到通过函数指针进行间接调用的语句时，根据函数指针的指向信息更新调用图，并在新的调用图上继续进行指向分析，如此迭代地更新调用图和在新调用图上进行指针分析，直至分析达到不动点。但是更新调用图的时机会影响指向分析的效率，尤其是在执行流敏感的指向分析时。

Emami 等[50]、Sotin 等[84]和 Horváth 等[85]在遍历调用图的过程中，如果遇到间接调用，就直接根据当前程序点处的函数指针指向更新调用图，立刻分析可能的被调用函数，最后合并每个被调用函数的输出指向集，就可以获得间接调用的指

向信息。在这之前,指针分析对函数指针的处理大多是根据整个程序中/某个过程中函数指针所有的可能指向来构建调用图,然后在调用图的基础上进行指针分析,而不会在指针分析过程中根据调用点处的实际指向逐一地更新调用图。

Hind 等[86]提出了一个能够处理函数指针的过程别名分析框架,其按照拓扑排序顺序遍历完初始调用图中的每个函数后,再根据遍历过程中收集到的所有函数指针别名信息更新调用图,并在新的调用图基础上迭代地更新别名信息,直至别名结果收敛。

Atkinson 等[87]利用函数类型签名对函数指针指向集进行过滤,从而得到更加准确的指向信息,但是这种类型过滤算法是在指针分析过程执行完之后进行的。Sun 等[88]则结合了文献[53]和文献[87]的方法,在遇到使用函数指针进行的函数调用时就进行类型过滤,这样可以在减少函数指针指向集的同时提高指针分析效率。

上述方法都是按照拓扑排序顺序遍历调用图,但实际上,按照逆拓扑排序顺序遍历调用图更加常见。Cheng 等[89]在过程间阶段,沿着调用图自底向上地根据传递函数进行过程间分析,由于函数指针的存在,还需要沿着部分调用图执行自上而下的分析,因此过程间分析阶段会迭代地执行自下而上和自上而下的传播,直到达到不动点。

3.2.3 其他消除虚假调用关系的方法

除了上述同时进行指针分析和构建函数调用图的方法外,还有一些方法可以避免产生虚假的调用关系。Wilson 等[53]提出了部分传递函数(Partial Transfer Function,PTF)来描述过程行为。具体地,只为特定的别名模式(不需要汇总所有可能的别名)总结一个过程,只要过程输入的别名相同,就在相关的调用上下文中重用该 PTF。部分传递函数中会记录可能的指针值,而函数指针的值会以扩展参数的形式表示。要解决通过函数指针进行间接调用的问题,只需要将 PTF 中的这些扩展参数标记为函数指针并记录它们的值即可。

函数指针分析[90]是近年来新提出的概念,作为指针分析的特例,函数指针分析缩小了需要分析的指针的范围,即限制为函数指针。函数指针分析在过程内阶段仅分析与函数指针相关的指令,而过程间阶段只会合并参数和返回值中包含函数指针的上下文信息。该文提出了一种强连通分量级别的算法,整个方法包括 3 个阶段:第一阶段获取符合定义的全局变量;第二阶段分析等价类并且构建函数指针图(FPG);第三阶段按照逆拓扑排序顺序遍历函数指针图,根据函数摘要的状态决定是否合并上下文信息,通过不断迭代得到最终结果。

FA 指针分析[91]是基于程序分解来构建包含函数指针的调用图。Zhang 等[92]对 FA 指针分析进行了实验,结果表明该方法计算的结果足以构建精确的程序调用图。

4 保守指向信息的消除

指针分析过程中,消除因流敏感分析不准确导致的保守指向信息,可以有效提高流敏感指针分析的精度,从而提高软件可信性。

在流敏感的指针分析中,堆指针和域敏感性[75,93]对执行强更新来说都是极大的挑战,都涉及如何对程序空间进行抽象。对程序空间进行抽象是为了能够更加准确地描述一个变量/域在一个程序或一个过程内唯一的位置,从而在流敏感强更新的过程中能与其他变量/域进行区分。

除了对栈空间、堆空间和数组/结构体元素进行抽象外,还有一些文章对指针分析的强更新进行了优化。这些方法与上述方法的不同之处在于其不聚焦于变量/域的唯一性,而是对于强更新算法运行过程中的精度/效率损失问题提出了解决方案。

4.1 问题描述

强更新是提高流敏感指针分析精度的最有效的方法之一,这种方法通过流敏感的指针分析获取指针在运行中对应的准确的内存位置,并随程序语句的执行用新值完全覆盖目标对象之前的内容,可以消除流不敏感的分析中不确定的指向(保守的指向信息)。图 4 给出了示例代码,在第 4 行时程序令指针 a 指向 b,在第 5 行时令指针 a 指向 c。如果是无法执行强更新的流不敏感算法,只能得到 $pts(a)=\{b,c\}$;流敏感的算法在第 4 行处可以得到 $pts(a)=\{b\}$,因此在第 5 行可以对变量 a 执行强更新,将 a 指向集中原有的内容完全覆盖,得到 $pts(a)=\{c\}$。

1	$x=\&a$
2	$y=\&b$
3	$z=\&c$
4	$*x=y$
5	$*x=z$

图 4 强更新示例程序

Fig. 4 Example of program of strong update

4.2 消除保守指向信息的方法

4.2.1 对程序运行时空间进行抽象

程序中的栈空间和堆空间都是动态分配的,一个变量声明对应的存储空间个数和一个存储空间分配(new)语句分配的对象个数都具有不确定的上限。为有效地进行程序分析,必须将这些运行时空间抽象到有限的范围内。抽象出来的结果通常称为抽象空间,包括抽象栈空间和抽象堆空间。

栈空间抽象较为简单,通常用一个或多个抽象栈空间描述某个有名变量。在同一个调用上下文中,一个过程内的同一个抽象栈空间一般表示唯一的物理空间。

Emami 等[48-50]使用抽象栈位置之间的指向信息作为数据流事实,并将指向关系标记为 may 和 must 以执行间接赋值的强更新。

Wilson 等[53]将内存划分为连续的块,并引入扩展参数来实现过程间的强更新。指针在调用上下文中具有许多可能的指向,但是指针在一个程序点只能包含其中一种可能性,因此扩展参数是该过程范围内的唯一块。仅当多个位置指向扩展参数并且该参数的实际值不是唯一位置时,才必须将该参数标记为不唯一。这种做法可以大大提高指针分析的强更新能力。该文定义了一个新的抽象——位置集(Location Set),以指定内存块和该块中的一组位置。在大多数情况下,此低级表示形式提供的结果可与基于高级类型的结果相媲美。但是这种方法比较保守,在某些情况下精度可能不够高。

而堆空间分配比栈空间分配具有更强的动态性，其抽象也更为复杂。通常的做法是将某条语句上动态分配的物理空间表达为一个抽象空间。堆空间抽象中最大的问题是即使在某一指定上下文中，一个抽象堆空间也能够表示多个不同的物理空间，这给空间的区分和表达式间关系的分析带来了很大困难。

对堆内存位置执行强更新的问题在 Java 程序中尤其突出，Java 的对象都通过 new 指令建立，并在堆上开辟一段内存，在对这些对象进行指向分析时，需要用一个抽象对象来表示在该分配点上分配的所有具体对象，因此指向图中节点和对象可能会存在一对多或是多对一的关系。如果要执行强更新，首先就需要解决节点和对象的对应问题。在 Java 程序中有两种常见的对象表示方法：同一个类的所有对象实例使用一个抽象的对象来表示[94-96]以及用对象分配点（Allocation Site）来区分同一类的不同对象的实例[97-98]。

Balatsouras 等[99]根据抽象对象的类型在分配点上是否可用来创建抽象对象，若抽象堆下给的类型在分配点上可用，则直接记录该类型；若不可用，则为每个分配点创建多个抽象对象，也就是对象可能有多个类型，每个类型对应一个抽象对象，这样可以有效获取堆对象的结构信息。

Fink 等[100-101]使用唯一性分析（Uniqueness Analysis）来识别代表单个实时具体对象的抽象对象，以便对这些对象执行强更新。

4.2.2 对数组/结构体进行抽象

此外，域敏感性则涉及对数组/结构体的抽象，域敏感性指分析过程中是否对对象的不同域成员进行描述以区分对象域的指向关系。域不敏感的算法将域的指向关系与其所在的对象/结构体统一处理，而域敏感的算法则将对象的域成员看作独立的数据结构。

域敏感的指针分析最常用的是基于访问路径分析（Access Path Analysis）的方法，访问路径指后跟零个或多个访问字段的局部变量，给定程序 P 中变量集合 V 和域集合 F，访问路径指集合 $V(.F)*$。利用访问路径分析可以很好地区分数组/结构体的元素，实现域敏感分析，但是这种方法可能存在无限访问路径的问题。

Landi 等[64]使用访问路径表示对象，并利用 k-limiting 技术来限制递归定义数据结构（如链表）。

De 等[66]针对无限访问路径提出了两种优化方法，一是使用部分 SSA 形式对程序进行转换，这样可以在每个函数中维护从局部变量到它们的指向集的单映射；二是基于访问路径的指向信息通常只在它所在的程序语句中使用的事实，文中只维护实时访问路径到它们的指向集的映射。这两种优化方法可以有效减少存储在每个程序语句中的映射的大小。

Prabhu 等[16]对 SSA 形式进行了扩展，提出了 Heap Array SSA，可以对 Java 的对象域成员进行 SSA 转换以实现域敏感分析。

4.2.3 其他

Lhoták 等[102]提出了混合强更新分析算法，结合流敏感和流不敏感这两种算法进行分析。当一个强更新提高了给定指向集的精度时，更精确的指向集可能会导致进一步的强更新，因此，为了精确地进行整体分析，必须精确地对这些小集合进行建模。单例集的表示和传播不会消耗太多的时间和空间，因此，对单例集进行流敏感传播，不会显著增加原本对流不敏感分析的复杂度或运行时间。该文扩展了单例集的流敏感建模，以启用强更新，为所有的指针维护了流不敏感的指向集，并为这些单例的程序点提供了流敏感的指向集。当流敏感集可用时，分析将优先使用流敏感集，以执行强更新；当没有可用的流敏感集时，则使用流不敏感的信息。

Sui 等[103]实现了按需强更新分析，在混合多阶段分析的框架中，通过预计算的不精确的指针值精确地执行强更新。算法分为两个阶段，分别是流敏感上下文敏感分析和流敏感分析。如果在预算范围内没有回答查询，就会在下一阶段将自己降级为更具可扩展性但精确度更低的分析。

5 其他指针分析精度优化问题

5.1 递归调用

存在递归调用的情况下，指针分析不能静态地获取准确的程序调用结构，函数调用图也不能仅通过深度遍历程序的调用结构来构建。因此，处理递归调用也涉及同时进行指针分析和构建函数调用图的问题，而且在过程间流敏感指针分析中，递归调用可能会使算法无法停止。

Emami 等[48-50]根据调用链上的函数名称匹配来解决上述问题，如果叶子节点函数名称与从 main 开始的调用链上某一个祖先节点的函数名称相同，则终止深度优先遍历，并将祖先节点标记为递归节点，叶子节点标记为近似节点。在指针分析过程中不需要分析近似节点，只使用具有相应输入的递归节点的输出作为结果。

Wilson 等[53]使用调用栈来记录调用上下文，因此可以通过后向搜索来查看调用目标是否在调用栈上以检测递归调用，然后根据调用栈中已经存在的 PTF 的摘要迭代地计算该递归调用的摘要。并且由于递归循环入口处的 PTF 是多个调用上下文的近似值，该文还为递归 PTF 记录了两个单独的输入域，一个指定递归循环外部调用的原始输入域，另一个保存所有递归调用的输入。

Vivien 等[104]在逃逸分析中不检查递归，原因是如果节点逃逸到递归方法中，那么从理论上来说分析永远不会终止。

Rugina 等[105]也使用调用栈来记录调用上下文。如果分析到递归函数，那么算法不会重新分析该过程，而是会使用当前的最佳分析结果并记录依赖于该结果的分析。如果最佳结果发生变化，算法将重复所有相关的分析，直到达到不动点。

RCI[71]是一种模块化指针分析，它将程序分成一个个模块进行指针分析，这里的模块就是调用图中的强连通分量 SCC。但 RCI 是通过一种线性时间的分析算法来识别 SCC，此处不过多赘述。

5.2 并发程序的指针分析

对并发程序进行指针分析的难度在于，并发程序在运行时，多个任务之间的交替运行空间巨大，各个线程之间也会交替执行，在分析指针的指向时也会受到线程之间的干扰，导致计算结果不准确。

Rugina 等[51,105]针对并发线程之间的干扰提出了流敏感

的指针分析算法，该算法旨在分析具有结构化并发构造的进程，包括 fork-join 结构、并行循环和有条件生成的线程；对于同步构造，如信号量、锁等的进程只进行保守的分析。为了避免线程之间互相干扰，算法会额外生成一个指向图来提取干扰信息，即反映并行线程中的指针分配语句对指针指向的影响，在计算每个语句对下一个程序点指向图的影响时，将根据这些干扰信息对指向关系进行调整。

Yu 等[106]为基于 CFL 可达性的按需指针分析引入了一种解决方案，使用数据共享和查询调度这两种技术来避免冗余的图遍历。通过数据共享实现并行化，在回答并发查询时发现的路径将被记录，后续查询可以根据记录重用，而不是重新遍历其关联的路径。通过查询调度，根据静态估计的依赖关系对查询进行优先级排序，从而进一步避免更多的冗余遍历。

Yu 等[107]提出了一种名为 Parseeker 的可扩展方法，利用 CFL 可达性并行化流敏感的需求驱动的指向分析，以提高指向分析的精度。该方法的主要特点在于：为目标进程生成和处理一组细粒度、可并行的指向关系查询；采用基于 CFL 可达性的指向分析来回答每个查询；同时加入 Map-Reduce 用于并行化查询，并设计了 3 种优化策略以进一步提高效率。

Ngaraj 等[108]将分阶段的流敏感指针分析描述为图重写问题，确定了其中影响流敏感指针分析的两个来源，并根据其定义的图重写公式来提取并发线程的分析，这种算法可以扩展到 8 个线程。

Zhao 等[109]对 SSA 形式进行了修改，结合了指向关系和堆上的定义使用链，针对并发程序提出的挑战，将 worklist 处理分为 4 个阶段，每个阶段对应一个单独的操作，这样可以最大程度地减少同一阶段中元素之间共享数据所需要的同步时间。随着线程数量的增加，该算法的性能也会提升。

6 总结与展望

6.1 总结

过程间分析和流敏感分析是提高指针分析精度的两种方法，利用这两种方法可以得到更为准确的指向分析结果，这些结果可以被其他软件分析工具所使用。一方面，由于指向关系更准确，这些工具的输入数据会减少，从而提高了软件分析工具的效率；另一方面，更准确的指向关系也有利于进一步提升软件分析工具的准确性，从而提高软件的可信性。

本文主要聚焦过程间流敏感的指针分析是如何提高精度的，分别从消除虚假信息和消除保守的指向信息两个方面进行了讨论。表 1 列出了文中指针分析方法的优缺点。

表 1 过程间流敏感指针分析方法汇总

Table 1 Interprocedural flow-sensitive pointer anlaysis

精度优化问题	所用方法	方法概述	优点	缺点
消除虚假返回路径	基于函数摘要的方法	按顺序遍历函数调用图，对每个函数进行过程内分析，收集指向关系并生成摘要，再次遇到该函数的调用时，直接根据函数的入口信息和摘要得到输出结果	生成函数摘要并在后续调用时使用，无需重复分析函数，从而提高过程间分析效率	暂无通用的函数摘要表示方法，函数摘要的设计对过程间分析的精度和效率影响很大，如用输入输出状态对表示则有可能消耗极大内存
	基于调用串的方法	以调用栈对应的某个序列作为上下文，然后将过程在不同上下文中的实例当作不同实体来处理	在指向关系上附加调用串可以直接获得某调用上下文中的指向结果，提高效率	可能存在无限访问路径的问题，需要对调用串的长度/深度进行限制
消除虚假调用关系	同时进行指针分析和构建调用图	构建初始的(不完整的)函数调用图并进行指向分析，在遇到间接调用语句时，根据函数指针更新调用图，并在新的调用图上继续进行指向分析，直至完成分析	随指针分析的结果更新函数指针的指向，精度较高	随程序规模增大，更新调用图的时机和次数会影响分析的效率
消除保守指向关系	对程序运行空间抽象	程序运行空间分为栈空间和堆空间，变量声明对应的存储空间个数和存储空间分配(new)语句分配的对象个数上限均不确定，须将这些空间抽象到有限的范围内	对程序运行空间进行抽象可以有效分辨变量和分配对象的数量，从而准确识别指向关系	一个抽象堆空间能够表示多个不同的物理空间，堆空间和物理空间之间的映射关系极大地影响了精度
	对数组/结构体抽象	指针分析过程中，针对对象的不同域成员进行抽象，从而区分不同对象的不同域成员的指向关系	用访问路径($V.(F)*$的形式)表示局部变量及其域成员，可以很方便地区分数组和结构体	可能存在无限访问路径的问题，需要对域成员的数量做限制

6.2 挑战与展望

指针分析一直是程序分析领域的热点研究问题，一是因为程序设计语言中指针的使用较为频繁，且会使程序变得复杂，需要对其进行分析，后续的程序优化、程序理解等工作才能得以顺利进行；二是因为指针分析结果的精确性会极大地影响程序分析工具的效率和精确性。

近年来，面对大规模程序的过程间流敏感指针分析，许多学者提出了很多方法来进一步提高精度/效率。大规模程序的过程间分析不仅需要记录大量的过程分析结果，还需要考虑文件间的指针信息交互；同时，随着程序规模增大，如果进行完全的指针分析，消耗的时间和空间也会呈指数增长，此时则需要考虑如何在有限时间内为其他软件分析工具提供准确的指向信息。

我们认为，过程间流敏感指针分析潜在的研究方向包括以下几个方面。

1)选择性过程间流敏感指针分析。从提高整个软件/程序的过程间指针分析效率来考虑，可以对部分过程进行区分调用上下文的分析。对于部分过程来说，过程/方法内部的指针指向及其改变与调用上下文无关，因此，是否区分调用上下文并不会影响指针分析的结果，对这部分过程/方法进行不区分调用上下文的分析可以进一步提高分析效率。目前已有部分文献通过机器学习[25]、模式匹配[26,110]、CFL 可达性[27,111-112]等方法筛选出需要进行区分调用上下文的过程，有效提高了分析效率。未来我们还可以结合其他领域的方法提出更多筛选的条件，进一步提高筛选精度。

流敏感指针分析相对于流不敏感的分析来说，精度有所

提升，但效率低下，而随着学者们对流不敏感分析的改进，流敏感指针分析的精度提升越来越有限。甚至有文献指出，对于某些测试程序(部分指针指向在过程中几乎不发生改变)，流敏感和流不敏感分析的精度没有区别。那么我们也可以考虑对程序中随数据流改变的指针选择性地进行流敏感的分析。

2)结合函数式语言特性进行动态过程间流敏感指针分析。为了进一步提高指针分析的精度，可能会根据指定的测试用例来运行程序，并根据程序运行的实际过程来进行动态指针分析，这样也可以更好地处理编程语言中的动态属性。动态指针分析最大的问题在于分析所需的代价很大，但Mock等[113]通过实验证明动态的指向集平均大小约为1，即只要分析足够精确，所需的内存可能不会很大。

函数式语言近年来受到越来越多的关注，由于语法简洁，功能强大，函数式语言的部分特性被新兴语言借鉴，用于大数据分析等领域。鉴于此，我们也可以借鉴函数式语言的相关特性来实现动态指针分析。例如，高阶函数特性(函数可以作为其他函数的参数)可以结合基于函数摘要的方法，将函数的分析结果保存为高阶函数式的指向摘要，在过程间分析时，按照形参实参的对应关系，将高阶函数式的指向摘要在函数调用点处实例化，从而避免调用上下文问题；同样地，高阶函数也可以与动态指针分析结合，将变量设为高阶函数的参数，分析过程中可根据不同的输入值更改分析结果。相比普通的动态指针分析，运用高阶函数特性的动态指针分析效率会进一步提升。此外CPS风格、列表内涵特性等同样也可以应用于指针分析领域，以提高分析效率。

参 考 文 献

[1] HIND M, PIOLI A. Which Pointer Analysis Should I Use? [J]. ACM Sigsoft Software Engineering Notes, 2000, 25(5): 113-123.

[2] STEENSGAARD B. Points-to Analysis in Almost Linear Time [C]// Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL '96). Association for Computing Machinery, 1996: 32-41.

[3] ANDERSEN L O. Program Analysis and Specialization for the C Programming Language[D]. Denmark: University of Copenhagen, 1994.

[4] BURKE M, CARINI P, CHOI J D, et al. Flow-insensitive interprocedural alias analysis in the presence of pointers[C]// Proceedings from the 7th Workshop on Languages and Compilers for Parallel Computing. Berlin: Springer, 1995: 234-250.

[5] CHOI J D, BURKE M G, CARINI P R. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects[C]// 20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages. ACM, 1993: 232-245.

[6] HIND M. Pointer Analysis: Haven't We Solved This Problem Yet? [C]// Acm Sigplan-sigsoft Workshop on Program Analysis for Software Tools & Engineering. ACM, 2001: 113-123.

[7] CHEN C M, HUO W, YU H T, et al. A Review of Pointer Analysis Optimization Techniques Based on Inclusion[J]. Chinese Journal of Computers, 2011, 34(7): 1224-1238.

[8] YAO Y F. A Review of Trusted Pointer Analysis Techniques For Software Trust- worthiness [J]. Applied Research of Computers, 2012, 29(2): 427-431.

[9] LI H F, MENG H N, ZHENG H J, et al. Research on Context-Sensitive Pointer Analysis of Object-Oriented Programs [J]. Journal of Software, 2022, 33(1): 78-101.

[10] HORWITZ S. Precise flow-insensitive may-alias analysis is NP-hard[J]. Acm Transactions on Programming Languages & Systems, 1997, 19(1): 1-6.

[11] LANDI W. Undecidability of static analysis[J]. ACM Letters on Programming Languages and Systems, 1996, 1(4): 323-337.

[12] RAMALINGAM G. The undecidability of aliasing [J]. ACM Transactions on Programming Languages & Systems, 1994, 16(5): 1467-1471.

[13] LI Q. Research on points-to analysis for Java[D]. Nanjing: Nanjing University, 2012.

[14] ANKUSH P, VAIBHAV B, SORAV B. OOElala: order-of-evaluation based alias analysis for compiler optimization[C]// 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, 2020: 839-853.

[15] WHALEY J, RINARD M. Compositional pointer and escape analysis for Java programs[C]// Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99). Association for Computing Machinery, 1999: 187-206.

[16] PRABHU P, SHANKAR P. Field Flow Sensitive Pointer and Escape Analysis for Java Using Heap Array SSA[C]// International Static Analysis Symposium(SAS 2008). Berlin: Springer, 2008: 110-127.

[17] TOK T B, GUYER S Z, LIN C. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers[C]// Lecture Notes in Computer Science. Berlin: Springer, 2006: 17-31.

[18] WHALEY J, MONICA S. LAM. An Efficient Inclusion-Based Points-To Analysis for Strictly-Typed Languages[C]// International Static Analysis Symposium (SAS 2002). New York: ACM, 2002: 180-195.

[19] REIF J H, Lewis H R. Symbolic evaluation and the global value graph[C]// Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. New York: ACM, 1977: 104-118.

[20] HARDEKOPF B, LIN C. Semi-Sparse Flow-Sensitive Pointer Analysis[C]// Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Association for Computing Machinery, 2009: 226-238.

[21] HARDEKOPF B, LIN C. Flow-sensitive pointer analysis for millions of lines of code[C]// International Symposium on Code Generation and Optimization. Piscataway, NJ: IEEE, 2011: 289-298.

[22] SUI Y, XUE J. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++[J]. IEEE Transactions on Software Engineering, 2018, 46(8): 812-835.

[23] LANDI A, BRUNSWICK S N, RYDER B G, et al. Interproce-

dural Aliasing In The Presence Of Pointers[D]. New Jersey: Rutgers University, 1993.

[24] SAMUEL Z G. Incorporating domain-specific information into the compilation process[D]. Austin: The University of Texas, 2003.

[25] JEONG S, JEON M, CHA S, et al. Data-driven context-sensitivity for points-to analysis[C]// Proceedings of the ACM on Programming Languages. ACM, 2017: 1-28.

[26] LI Y, TAN T, MOLLER A, et al. A Principled Approach to Selective Context Sensitivity for Pointer Analysis[J]. ACM Transactions on Programming Languages and Systems, 2020, 42(2): 1-40.

[27] LU J, HE D, XUE J. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity[J]. ACM Transactions on Software Engineering and Methodology(TOSEM), 2021, 30(4): 1-46.

[28] LU J, HE D, XUE J. Selective context-sensitivity for k-CFA with CFL-reachability[C]// International Static Analysis Symposium(SAS 2021). Springer. 2021: 261-285.

[29] MILANOVA A, ROUNTEV A, RYDER B G. Parameterized object sensitivity for points-to analysis for Java[J]. ACM Transactions on Software Engineering & Methodology, 2005, 14(1): 1-41.

[30] KASTRINIS G, SMARAGDAKIS Y. Hybrid context-sensitivity for points-to analysis[C]// the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 13). ACM, 2013: 423-434.

[31] LHOTáK O, HENDREN L. Context-Sensitive Points-to Analysis: Is It Worth It? [C]// International Conference on Compiler Construction(CC 2006). Springer, 2006: 47-64.

[32] NAIK M, AIKEN A, WHALEY J. Effective static race detection for Java[C]// Acm Sigplan Conference on Programming Language Design & Implementation. ACM, 2006.

[33] SRIDHARAN M, CHANDRA S, DOLBY J, et al. Alias Analysis for Object-Oriented Programs[M]. Lecture Notes in Computer Science. Berlin: Springer, 2013: 196-232.

[34] TAN T, LI Y, XUE J. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting[C]// International Static Analysis Symposium(SAS 2016). New York: ACM, 2016: 489-510.

[35] TIAN T, YUE L, XUE J. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata[C]// Acm Sigplan Conference on Programming Language Design & Implementation. ACM, 2017: 278-291.

[36] SMARAGDAKIS Y, BRAVENBOER M, LHOTÁK O. Pick your contexts well: Understanding object-sensitivity[J]. ACM SIGPLAN Notices, 2011, 46(1): 17-30.

[37] LUNDBERG J, GUTZMANN T, EDVINSSON M, et al. Fast and precise points-to analysis[J]. Information and Software Technology, 2009, 51(10): 1428-1439.

[38] BAO Y, ZHANG C, ZHUO X, et al. Parameter Sensitive Pointer Analysis for Java[C]// 26th International Conference on Engineering of Complex Computer Systems(ICECCS). IEEE, 2022: 162-167.

[39] PEARCE D J, KELLY P, HANKIN C. Efficient field-sensitive pointer analysis of C[J]. ACM Transactions on Programming Languages and Systems, 2007, 30(1): 1-42.

[40] ANTOINE M. Field-Sensitive Value Analysis of Embedded C Programs with Union Types and Pointer Arithmetics[C]// ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems. Ecole Normale Superieure, 2006: 54-63.

[41] SUI Y, FAN X, ZHOU H, et al. Loop-oriented array- and field-sensitive pointer analysis for automatic SIMD vectorization. [J]. ACM SIGPLAN Notices, 2016, 51(5): 41-51.

[42] LATTNER C, LENHARTH A, ADVE V. Making context-sensitive points-to analysis with heap cloning practical for the real world[C]// Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI' 07). ACM, 2007: 278-289.

[43] SUI Y, YE S, XUE J, et al. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA[C]// Proceedings of the 9th Asian Conference on Programming Languages and Systems (APLAS'11). ACM, 2011: 155-171.

[44] WEI S, RYDER B. State-Sensitive Points-to Analysis for the Dynamic Behavior of JavaScript Objects[C]// Proceedings of the 28th European Conference on ECOOP 2014. Springer-Verlag, 2014: 1-26.

[45] ANTONIOL G, CALZOLARI F, TONELLA P. Impact of Function Pointers on the Call Graph[C]// European Conference on Software Maintenance & Reengineering. IEEE Computer Society, 1999: 51-59.

[46] CHENG B, HWU W. An Empirical Study of Function Pointers Using SPEC Benchmarks[C]// Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing. Berlin: Springer, 1999: 490-493.

[47] MILANOVA A, ROUNTEV A, RYDER B G. Precise Call Graph Construction in the Presence of Function Pointers[C]// IEEE International Workshop on Source Code Analysis & Manipulation. IEEE, 2003: 155-162.

[48] FHNDRICH M, REHOF J, DAS M. Scalable context-sensitive flow analysis using instantiation constraints[J]. ACM SIGPLAN Notices, 2000, 35(5): 253-263.

[49] EMAMI M. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler[D]. Montreal: McGill University, 1993.

[50] EMAMI M, GHIYA R, HENDREN L J. Context-sensitive interprocedural points-to analysis in the presence of function pointers[J]. ACM SIGPHLAN Notices, 1994, 29(6): 242-256.

[51] RUGINA R, RINARD M. Pointer Analysis for Multithreaded Programs[J]. ACM SIGPLAN Notices, 2002, 34(5): 77-90.

[52] LIU Q. Research on Interprocess Analysis Technique Considering Pointer Alias[D]. Beijing: Chinese Academy of Sciences(Institute of Computing Technology), 1998.

[53] WILSON R P, LAM M S. Efficient Context-Sensitive Pointer Analysis for C Programs[J]. ACM SIGPLAN Notices, 1995, 30(6): 1-12.

[54] CHEN P S, HWANG Y S, JU D C, et al. Interprocedural Proba-

bilistic Pointer Analysis[J]. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(10): 893-907.

[55] QIAN L, ZHAO J, LI X. Optimize Context-Sensitive Andersen-Style Points-To Analysis by Method Summarization and Cycle-Elimination[C]// International Conference on Leveraging Applications of Formal Methods. Springer-Verlag, 2010: 564-578.

[56] DAVID T, TIMOTEJ K, NOAM R, et al. Past-Sensitive Pointer Analysis for Symbolic Execution[C]// 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering(ESEC/FSE'20). Association for Computing Machinery. 2020: 197-208.

[57] SCHUBERT P, HERMANN B, BODDEN E. Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis[C]// 35th European Conference on Object-Oriented Programming(ECOOP 2021). Schloss Dagstuhl, 2021: 1-31.

[58] WANG Y, ZHOU M, GU M, et al. Necessity and Capability of Flow, Context, Field and Quasi Path Sensitive Points-to Analysis[C]// 2019 26th Asia-Pacific Software Engineering Conference(APSEC). IEEE Computer Society, 2019: 268-275.

[59] SHARIR M, PNUELI A. Two approaches to interprocedural data flow analysis[D]. New York: New York University, 1978.

[60] SHIVERS O. Control-Flow Analysis of Higher-Order Languages[D]. Pennsylvania: Carnegie Mellon University, 1991.

[61] PALSBERG J, SCHWARTZBACH M I. Object-Oriented Type Inference[J]. ACM SIGPLAN Notices, 1997, 26(11): 146-161.

[62] GROVE D, DEFOUW G, DEAN J, et al. Call Graph Construction in Object-Oriented Languages[J]. ACM SIGPLAN Notices, 1997, 32(10): 108-124.

[63] JONES N D, MUCHNICK S S. Flow analysis and optimization of LISP-like structures[C]// Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. ACM, 1979: 244-256.

[64] LANDI W, RYDER B G. A safe approximate algorithm for interprocedural aliasing[J]. ACM SIGPLAN Notices, 1992, 39(4): 235-248.

[65] DEUTSCH A. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting[J]. ACM SIGPLAN Notices, 1994, 29(6): 230-241.

[66] DE A, D'SOUZA D. Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates[C]// European Conference on Object-Oriented Programming. Berlin: Springer, 2012: 665-687.

[67] SMARAGDAKIS Y, KASTRINIS G. Defensive Points-To Analysis: Effective Soundness via Laziness[C]// 32nd European Conference on Object-Oriented Programming(ECOOP 2018). Schloss Dagstuhl, 2018: 23: 1-23: 28.

[68] JEONG S, JEON M, CHA S, et al. Data-driven context-sensitivity for points-to analysis[C]// Proceedings of the ACM on Programming Languages. ACM, 2017: 1-28.

[69] HUA Y, SUI Y, CHEN S, et al. Spatio-Temporal Context Reduction: A Pointer-Analysis-Based Static Approach for Detecting Use-After-Free Vulnerabilities[C]// 2018 IEEE/ACM 40th International Conference on Software Engineering(ICSE). IEEE Computer Society, 2018: 327-337.

[70] AGESEN O. The cartesian product algorithm [C]// European Conference on Object-Oriented Programming (ECOOP 95). Schloss Dagstuhl, 1995: 2-26.

[71] GROVE D, CHAMBERS C. A Framework for Call Graph Construction Algorithms[J]. ACM Transactions on Programming Languages and Systems, 2001, 23(6): 685-746.

[72] CHEN T Q, FAN G S, YIN B H, et al. An Optimization Method for Interprocedural Analysis of Function Inlining Oriented to Abstract Interpretation Framework[J]. Journal of Software, 2022, 33(8): 2964-2979.

[73] WHALEY J, LAM M S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams[J]. ACM SIGPLAN Notices, 2004, 39(6): 131-144.

[74] THIESSEN R, LHOTÁK O. Context transformations for pointer analysis[J]. ACM SIGPLAN Notices, 2017, 52(6): 263-277.

[75] REPS T, HORWITZ S, SAGIV M. Precise interprocedural dataflow analysis via graph reachability[C]// ACM SIGPLAN-sigact Symposium on Principles of Programming Languages (POPL 95). ACM, 1995: 49-61.

[76] REPS T. Program analysis via graph reachability[J]. Information & Software Technology, 1998, 40(11/12): 701-726.

[77] JOHANNES S, LISA N Q D, ALI K, et al. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java [C]// 30th European Conference on Object-Oriented Programming(ECOOP 2016). Schloss Dagstuhl, 2016: 1-26.

[78] SUI Y, YE D, XUE J. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis[J]. IEEE Transactions on Software Engineering, 2014, 40(2): 107-122.

[79] SUI Y, PENG D, XUE J. Sparse flow-sensitive pointer analysis for multithreaded programs[C]// Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16). IEEE, 2016: 160-170.

[80] SUI Y, XUE J. SVF: interprocedural static value-flow analysis in LLVM[C]// Proceedings of the 25th International Conference on Compiler Construction(CC 2016). ACM, 2016: 265-266.

[81] SUI Y, XUE J. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++[J]. IEEE Transactions on Software Engineering, 2018, 46(8): 812-835.

[82] LEI Y, SUI Y. Fast and Precise Handling of Positive Weight Cycles for Field-Sensitive Pointer Analysis[C]// International Static Analysis Symposium (SAS 2019). Berlin: Springer. 2019: 27-47.

[83] SUN C, MIDKIFF S. Demand-Driven Refinement of Points-to Analysis[C]// 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 2019: 264-265.

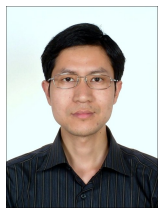
[84] SOTIN P, JEANNET B. Precise Interprocedural Analysis in the Presence of Pointers to the Stack[C]// Programming Languages and Systems. Berlin: Springer, 2012: 459-479.

[85] HORVÁTH E, FORGáCS I, KISS Á, et al. General flow-sensitive pointer analysis and call graph construction[C]// 9th Symposium on Programming Languages and Software Tools(SPLST 2005). 2005: 49-58.

[86] HIND M, BURKE M, CARINI P, et al. Interprocedural Pointer Alias Analysis[J]. ACM Transactions on Programming Langua-

ges & Systems,1999,21(4):848-894.
[87] ATKINSON D C. Accurate call graph extraction of programs with function pointers using type signatures[C]// ASIA-Pacific Software Engineering Conference. IEEE,2004:326-335.
[88] SUN H H,WANG S Y,LIN X B. Improvement of Interprocess Pointer Analysis Algorithm[J]. Computer Engineering,2009,35(1):90-92,104.
[89] CHENG B C,HWU W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation [J]. ACM Sigplan Notices,2000,35(5):57-69.
[90] ZHANG W,ZHANG Y. Lightweight function pointer analysis [C]// Information Security Practice and Experience (ISPEC 2015). Cham:Springer,2015:439-453.
[91] ZHANG X. Practical pointer aliasing analysis[D]. New Brunswick:Rutgers University,1998.
[92] ZHANG S,RYDER B G,LANDI W. Program Decomposition for Pointer Aliasing:A Step toward Practical Analyses[J]. ACM Sigsoft Software Engineering Notes,1996,21(6):81-92.
[93] YONG S,HORWITZ S,REPS T. Pointer analysis for programs with structures and casting[C]// Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI). ACM,New York,1999:91-103.
[94] BACON D F,SWEENEY P F. Fast Static Analysis of C++ Virtual Function Calls[J]. ACM SIGPLAN Notices,1996,31(10):324-341.
[95] DEAN J. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis[C]// European Conference on Object-Oriented Programming (ECOOP 1995). Berlin: Springer,1995:77-101.
[96] DIWAN A,MOSS J,MCKINLEY K S. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs[J]. ACM Sigplan Notices,1996,31(10):292-305.
[97] ROUNTEV A,MILANOVA A,RYDER B G. Points-to analysis for Java using annotated constraints[J]. ACM SIGPLAN Notices,2001,36(11):43-55.
[98] MILANOVA A,ROUNTEV A,RYDER B G. Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java[J]. ACM SIGSOFT Software Engineering Notes,2002,27(4):1-11.
[99] BALATSOURAS G,SMARAGDAKIS Y. Structure-Sensitive Points-To Analysis for C and C++[C]// International Static Analysis Symposium (SAS 2016). Berlin: Springer,2016:84-104.
[100] FINK S J,YAHAV E,DOR N,et al. Effective typestate verification in the presence of aliasing[J]. ACM Transactions on Software Engineering & Methodology,2006,17(2):133-144.
[101] FINK S,KNOBE K,SARKAR V. Unified Analysis of Array and Object References in Strongly Typed Languages[C]// International Symposium on Static Analysis. Springer-Verlag,2000:155-174.
[102] LHOTÁK O,CHUNG K C A. Points-to analysis with efficient strong updates[C]// ACM Sigplan-sigact Symposium on Principles of Programming Languages. ACM,2011:3-16.
[103] SUI Y,XUE J. On-demand strong update analysis via value-flow refinement[C]// ACM Sigsoft International Symposium on Foundations of Software Engineering. ACM,2016:460-473.
[104] VIVIEN F,RINARD M. Incrementalized pointer and escape analysis[J]. ACM SIGPLAN Notices,2001,36(5):35-46.
[105] RUGINA R,RINARD M C. Pointer analysis for structured parallel programs[J]. Acm Transactions on Programming Languages & Systems,2003,25(1):70-116.
[106] YU S,DING Y,XUE J. Parallel Pointer Analysis with CFL-Reachability[C]// 2014 43rd International Conference on Parallel Processing. IEEE,2014:451-460.
[107] YU H,SUN Q,XIAO K,et al,Parallelizing Flow-Sensitive Demand-Driven Points-to Analysis[C]// 2020 IEEE 20th International Conference on Software Quality,Reliability and Security Companion(QRS-C). IEEE,2020:91-97.
[108] NAGARAJ V,GOVINDARAJAN R. Parallel flow-sensitive pointer analysis by graph-rewriting[C]// 2013 22nd International Conference on Parallel Architectures and Compilation Techniques(PACT). IEEE,2013:19-28.
[109] ZHAO J,BURKE M G,SARKAR V. Parallel sparse flow-sensitive points-to analysis[C]// Proceedings of the 27th International Conference on Compiler Construction(CC 2018). ACM,2018:59-70.
[110] TAN T,LI Y,MA X,et al. Making pointer analysis more precise by unleashing the power of selective context sensitivity [C]// Proceedings of the ACM Programming Languages 5(OOPSLA). ACM,2021:1-27.
[111] HE D,LU J,GAO Y,et al. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability[C]// 35th European Conference on Object-Oriented Programming (ECOOP 2021). Dagstuhl,Germany. 2021.
[112] LU J,XUE J. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity[C]// Proceedings of the ACM on Programming Languages. ACM,2019:1-29.
[113] MOCK M,DAS M,CHAMBERS C,et al. Dynamic points-to sets:A comparison with static analyses and potential applications in program understanding and optimization[C]// ACM Sigplan-sigsoft Workshop on Program Analysis for Software Tools & Engineering. ACM,2001:66-72.

SHUAI Dongxin, born in 1997, postgraduate. Her main research interest is program analysis.



ZHANG Yingzhou, born in 1978, professor, is a senior member of China Computer Federation. His main research interests include software formal analysis, program slicing, service computing, functional programming technology.

(责任编辑:何杨)