软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2023,34(6):2606−2627 [doi: 10.13328/j.cnki.jos.006850]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向软件供应链的异常分析方法综述*

葛丽丽, 帅东昕, 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, 软件学院, 网络空间安全学院, 江苏 南京 210023)

通信作者: 张迎周, E-mail: zhangyz@njupt.edu.cn

摘 要: 软件在国民经济的各个领域占据越来越重要的地位. 万物互联的大背景下, 信息之间的交互、分析、协同变得越来越普遍, 程序/软件之间的依赖关系逐渐增多, 这使得人们对系统可靠性和健壮性提出了更高的要求. 由开源组件和第三方组件构成的软件供应链, 其所面临的安全问题近年来成为了学术界和工业界共同关注的焦点. 库函数作为开源软件的重要组成部分, 与软件供应链安全有着密切的联系. 为了提高软件开发效率, 软件库或应用程序编程接口(API)在程序编写过程中会被频繁使用, 但库函数中存在的错误或漏洞可能会被攻击者利用, 从而损害软件供应链安全. 这些错误或漏洞往往与库函数中存在的异常有关, 因此, 从精度和效率两方面对适用于库函数的异常分析方法进行总结归纳, 对于每种异常分析方法的基本思想和重要过程进行阐述, 并针对库函数异常分析面临的挑战给出初步解决思路. 对软件供应链中的库函数进行异常分析, 有助于增强软件系统的健壮性, 进而保障软件供应链的安全.

关键词: 软件供应链; 异常分析; 库函数; 精度优化; 效率优化; 函数摘要

中图法分类号: TP311

中文引用格式: 葛丽丽, 帅东昕, 谢金言, 张迎周, 薛渝川, 杨嘉毅, 密杰, 卢跃. 面向软件供应链的异常分析方法综述. 软件学报, 2023, 34(6): 2606–2627. http://www.jos.org.cn/1000-9825/6850.htm

英文引用格式: Ge LL, Shuai DX, Xie JY, Zhang YZ, Xue YC, Yang JY, Mi J, Lu Y. Review on Exception Analysis Methods for Software Supply Chain. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2606−2627 (in Chinese). http://www.jos.org.cn/1000-9825/6850.htm

Review on Exception Analysis Methods for Software Supply Chain

GE Li-Li, SHUAI Dong-Xin, XIE Jin-Yan, ZHANG Ying-Zhou, XUE Yu-Chuan, YANG Jia-Yi, MI Jie, LU Yue

(School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China)

Abstract: Software occupies an increasingly important position in various fields of the national economy. Under the background of the Internet of everything, interaction, analysis and collaboration of information are becoming more and more common, and dependencies among programs/softwares are increasing. It makes people put forward higher requirements for system reliability and robustness. A software supply chain consists of open source components and third-party components, and its security problems have become the focus of both academia and industry in recent years. As an important part of open source software, library functions are closely related to the security of the software supply chain. In order to improve software development efficiency, software libraries or application programming interfaces (APIs) will be frequently used in the process of programming, but errors or vulnerabilities in library functions may be exploited by attackers to compromise the security of the software supply chain. These errors or vulnerabilities are often related to exceptions in library functions. Therefore, the exception analysis methods of library functions are summarized from the two aspects of accuracy and efficiency in this study. The basic idea and important process of each exception analysis method are described, and a preliminary solution is given for the challenges faced by library function exception analysis. Exception analysis of library functions in the software supply chain is helpful to enhance the robustness of software system and to ensure the security of the software supply chain.

* 本文由“软件可信性与供应链安全前沿进展”专题特约编辑向剑文教授、郑征教授、申文博研究员、常瑞副教授、田聪教授推荐.

收稿时间: 2022-09-05; 修改时间: 2022-10-10, 2022-12-14; 采用时间: 2022-12-28; jos 在线出版时间: 2023-01-13

Key words: software supply chain; exception analysis; library function; precision optimization; efficiency optimization; function summary

在万物互联的大背景下, 信息之间的协同运作变得越来越普遍[1], 程序/软件之间的依赖关系越来越多, 系统可靠性和健壮性变得越来越重要. 复杂软件往往由开源软件组合而成, 这些开源软件彼此组合、依赖, 连同为各个开源软件做贡献的维护者和开发者, 共同构成了开源软件供应链. 异常[2]是程序执行过程中出现的不正常情况. 异常处理是一种功能强大且被广泛使用的编程语言抽象, 用于构建健壮的软件系统. 但是, 异常处理结构引入了一些额外的过程间控制流(异常控制流), 这使得异常处理代码本身更易产生缺陷/安全漏洞. 如果不能正确处理这些控制流, 可能导致安全漏洞、API 封装漏洞以及一定数量的安全策略违反等安全问题, 不法分子可以利用软件供应链中已有的安全漏洞进行组合攻击, 这使得软件供应链面临着安全风险.

库函数作为开源软件供应链[3]中的重要组成部分, 按照是否可以获取库函数的源代码, 可将软件供应链中的库函数分为三大类: 可获取源码(简称有源)的库函数、无法获取源码(简称无源)的库函数以及可获取部分源码(简称半有源)的库函数. 其中, 有源库函数主要包括各种程序中所调用的标准库函数和用户自定义的库函数, 无源库函数主要包括由外部程序提供的第三方库函数, 而有些标准库函数属于半有源库函数. 使用软件库或应用程序编程接口(API)是一种提高软件开发效率、减少软件开发成本的广泛方法[4], 但库函数中难免存在不合理的异常处理结构, 这将导致异常或错误得不到有效的处理. 如果没有对库函数中的异常/错误进行合理的处理, 将会出现系统崩溃等安全问题, 继而引发安全漏洞[5]. 例如 Xcode 非官方版本恶意代码污染事件的发生——通过 Core Service 库文件进行感染, 使得多个应用程序受到影响. 这起安全事件可能与库文件本身的健壮性有关, 因此对库函数中的异常/错误进行分析有助于程序开发人员获知异常所引起的控制流, 预判并修补安全漏洞, 有助于开发高质量的软件, 保证软件产品的安全可靠性, 增强软件系统的健壮性, 进一步保证软件供应链的安全.

广义上, 软件供应链[6]是指从软件供应商开始, 通过一级或多级软件设计、开发阶段编写软件, 借助软件交付渠道将软件传送至软件用户的系统. 狭义上, 软件供应链是指软件生产过程的供应链条, 包括由库函数组成的第三方组件、开发工具和开发环境等要素. 本文讨论的软件供应链指的是狭义的软件供应链, 主要关注软件供应链中的重要组成部分——库函数. 本文从保证软件系统本身的健壮性出发, 研究软件供应链安全相关问题, 这有别于已有的研究工作[5,7,8], 本文所讨论的异常主要是软件供应链库函数中引发的不正常/错误操作. 精度和效率是衡量各种异常分析方法好坏的主要方面.

- 精度主要以检出率和误报率为衡量指标: 检出率指的是通过各种方法进行分析后, 未捕获的异常个数占实际引发异常个数的百分比; 误报率指的是被测程序/库函数中某异常实际是未捕获的, 但是通过分析得出该异常已被捕获, 这种错误报告出的未捕获异常个数占实际未捕获异常个数的百分比;
- 效率主要体现在进行分析时所花费的开销, 即时间和空间开销上.

本文主要从精度和效率两方面对适用于库函数的异常分析方法进行总结、归纳, 包括精度优化和效率优化两部分. 这有助于提高软件系统的健壮性, 增强软件供应链的安全性.

本文第 1 节介绍异常相关的背景知识. 第 2 节主要从影响异常分析精度的方面——处理程序不可达的问题出发, 对适用于库函数中异常处理结构分析的精度优化方法进行归纳总结. 第 3 节对适用于库函数中异常处理结构分析的效率优化方法进行归纳总结. 第 4 节对软件供应链库函数异常分析方法进行分类总结, 并针对软件供应链库函数异常分析所面临的挑战给出我们初步的解决方案, 最后对未来的研究工作做出展望.

1 背景介绍

1.1 相关定义

- 异常(exception): 程序执行时遇到的非正常情况或者意外行为[9−13], 如代码或者调用的代码(如公共库)中有错误、操作系统资源不可用、公共语言运行库遇到意外情况(如无法验证代码)等. 常见的有数

组下标越界、算数溢出(超出数值表达范围)、除数为 0、无效参数等. 在这种情况下, 程序运行时本身可以解决, 由异常代码调整程序运行方向, 可使程序继续运行, 直至正常结束;

- 错误(error): 由系统错误或资源被占用导致的系统级问题, 最常见的错误有程序进入死循环、内存泄漏等. 在这种情况下, 程序运行时无法自行解决, 只能通过其他程序干预使程序能够继续运行, 但干预程序通常也不会对这类异常进行处理;
- 库函数(library function): 存放在函数库中的函数, 是开源软件供应链中的重要组成部分. 函数库是由系统建立的具有一定功能的函数的集合, 库中存放函数的名称和对应的目标代码以及连接过程中所需要的重定位信息. 库函数具有明确的功能、入口调用参数和返回值;
- API (application programming interface): 一些预先定义的函数, 目的是便于开发人员访问例程而又无需访问源码或理解内部工作机制的细节;
- 软件供应链: 从软件供应商开始, 通过一级或多级软件设计、开发阶段编写软件, 借助软件交付渠道将软件传送至软件用户的系统. 狭义上, 软件供应链是指软件生产过程的供应链条, 包括由库函数组成的第三方组件、开发工具和开发环境等要素.

1.2 程序异常相关概念

- 异常处理技术

异常处理是为响应并处理异常/错误而采取的后续操作[9,14]. 异常处理程序是与一个或多个异常相绑定的、在程序/过程中发生异常时执行的代码. 按照异常处理层次, 可以将异常处理技术分为软件级处理技术和硬件级处理技术, 本文主要关注软件级的异常处理技术. 这种异常处理技术可以分为基于返回值的技术和基于非返回值的技术: 基于返回值的技术是一种根据不同的返回值进行检查, 并对异常进行处理的技术, 例如 C 语言、Rust 语言、Go 语言, 都是通过返回值的方式来使调用者对异常/错误进行处理; 基于非返回值的技术可以分为基于编程语言的机制和基于操作系统的机制, 本文重点关注的是基于编程语言的机制, 也可称为异常处理机制, 主要可以分为类 raise-when/except[15]机制(例如 Ada 语言、Python 语言等)、类 throw-try-catch 机制[16−18](例如 Java、C++、C#、Scala 语言等)、类 setjmp-longjmp 机制(例如 C 语言)[12]. 具体分类如图 1 所示.

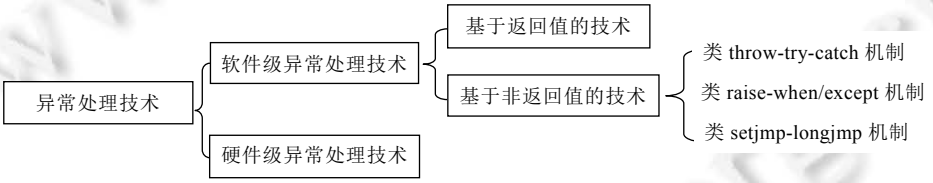

图 1 异常处理技术分类

- 异常流

异常流[19]指的是沿着异常处理路径所抛出的每个异常对象的流——从抛出语句到处理它的捕获块. 本文中的异常流指的是软件供应链中库函数异常对应的异常流, 往往需要进行一些特殊的处理.

(1) 异常的每一个定义和使用都与类型相关, 其中, 异常定义指的是异常引发, 异常的使用指的是异常捕获. 异常引发和捕获的类型需要匹配, 构成异常定义-使用链. 此外, 如果捕获到父类的异常, 子类的异常也会被捕获到;

(2) 数据流与程序执行流的方向相反, 因此, 异常流是一个反向的数据流问题;

(3) 与异常流相关的控制流语句是诸如 try、catch/except 块、throw/raise 等语句和方法调用语句, 其他语句不会影响异常流, 且方法中这些语句的顺序对程序中异常分析没有任何影响, 通常以调用图作为输入, 需要关注的重点是, throw、raise 或方法调用是否包含在嵌套的 try 块中. 为了计算方法中的异常流, 只需要关注从方法入口到每个 try-catch/except 或到不包含在任何 try-catch/except 块中的 throw/raise 或方法调用的路径.

- 异常处理过程

当程序中抛出异常时, 编程语言的异常处理机制负责将程序的正常控制流跳转至异常控制流. 因此, 当抛出异常时, 程序的正常操作将被中断, 并寻找适当的异常处理程序. 当找到适当的异常处理程序时, 执行它, 控制流将返回到处理程序后面的代码处[19]. 如果没有找到适当的异常处理程序, 所抛出的异常将被视为该过程未捕获的异常, 未捕获的异常将会沿着程序的执行流从被调用过程反向传播到调用过程中, 即发生异常传播. 下面以除数可能为 0 导致的 ArithmeticException/ZeroDivisionError 为示例, 说明异常处理过程, 如图 2 所示.

```
double divide(double from,double to){
from=20.0;
to=0;
int result;
if (to==0){
      printf (stderr,"除数为 0  退出运行...");
return −1;
}
result=from/to;
printf (stderr,"result 变量的值为: %d\n",result);
return result;
}
```

(a) 基于返回值的异常处理技术

```
def ThrowErr( ):
raise ZeroDivisionError ("抛出一个除数为 0 异常")
class MuffledCalculator:
muffled=false
def cala(self,expr):
try:
   return eval(expr)
except ZeroDivisionError:
   if self.muffled:
      print "除数不能为 0!"
```

(b) 类 raise-when/except 机制

```
double divide(int a,int b)
{
   if(b==0)
   {
      throw Div_by_zero_Exception();
   }
   return (a/b);
}
 void foo( )
{
   int x=20;
   int y=0;
   try {
      double z=divide(x,y);
      cout <<z<<endl;
   }
   catch (Div_by_zero_Exception e) {
      cout<<"除数为 0！"<<endl;
   }
}
```

(c) 类 throw-try-catch 机制

```
static jmp_buf env;
double divide(double to,double by){
if (by==0){
   longjmp(env,1);
}
   return to/by;
}
void foo( ){
if (setjmp(env)==0){
      divide(20,0);
}
else{
      printf ("除数不能为 0!");
}
printf ("完成!");
}
```

(d) 类 setjmp-longjmp 机制

图 2 异常处理过程示例

- 异常分析

异常分析[13]指的是编程语言与软件工程之间的一个跨学科领域. 如果没有通过异常分析获得任何信息, 编程语言和软件工程中的任务将过于保守, 或者可能会错过异常情况. 通过考虑所有可能的执行程序, 这些任务可以变得更健全或更复杂. 自从在编程语言中引入异常处理以来, 已经出现了各种异常分析, 它们可以静态或动态地分析程序的异常行为, 即: 通过对异常处理代码进行分析, 可以获知程序在发生异常时的执行路径——从抛出异常到捕获异常、异常变量的传播路径、该程序执行过程中异常的控制流、该过程中未捕获的异常, 这些都是异常分析的主要内容. 本文中的异常分析指的是对软件供应链中库函数的异常处理结构进行分析.

1.3 研究动因及相关工作

软件供应链的开源化, 使得供应链安全不可避免地受到软件系统自身安全性的影响, 如果在软件供应链

的某一环节使用了含有安全缺陷的开源软件或者在开源软件中使用了带有安全缺陷的组件, 这将会给软件供应链的相关环节带来不可消除的安全隐患, 继而对软件供应链的安全产生威胁. 由于软件系统的安全性与其健壮性密不可分, 在一些编程语言中使用异常处理机制来提高软件健壮性, 但异常处理机制的不恰当使用, 可能会引发一些安全缺陷或漏洞. 例如: 当使用所有异常的父类 Exception 类对程序中的特殊异常进行处理时, 虽然可以使捕获块变得简洁, 但是由于捕获的异常过于广泛, 当在程序中抛出某一个新类型的具体异常时, 由于 Exception 屏蔽了该具体异常, 将会使得该具体异常得不到有效的捕获, 为了捕获它, 会使异常处理代码变得非常复杂, 更可能引发安全缺陷——通用异常捕获声明缺陷(CWE-396: Declaration of catch for generic exception)[20]. 软件供应链中往往存在复杂的软件链条, 库函数作为软件系统中的重要组件, 如果库函数中的异常没有得到正确而有效的处理, 将会产生一些安全漏洞, 造成安全风险. 所以, 本文从对软件系统自身安全性出发, 对库函数中的异常处理机制进行分析, 研究软件供应链安全相关的问题, 这对提升软件供应链的整体安全具有一定的意义.

近年来, 软件供应链安全受到国内外各界的关注. 一些学者从不同角度对软件供应链的相关研究进行了综述. 何熙巽等人[6]从软件供应链的定义和发展历程出发, 介绍了软件供应链的两个方面的安全问题, 并对软件供应链安全的研究现状进行了分析总结. 崔宝江[21]从技术层面提出了加强软件供应链安全的措施, 可以有效防止安全缺陷给软件供应链带来的威胁. 由悬镜安全制作的软件供应链安全白皮书[22]通过对软件供应链的安全现状进行梳理、剖析, 提出了对软件供应链的安全风险进行防范与治理的举措. 缪尚廷在文献[5]中提出对第三方组件进行独立的测试和分析, 掌握开源软件组件隐藏的安全漏洞或缺陷, 可以有效保障开源软件供应链的安全. 以上综述文章虽然对于软件供应链安全所面临的挑战及应对供应链攻击的策略进行了分析和总结, 但是都没有从软件系统本身的健壮性出发, 深入研究软件供应链安全的相关问题.

由于国内外学者对程序健壮性的研究逐渐增多, 异常分析作为提高软件健壮性的一种有效方式取得了一定的进展, 有一些针对特定类型语言/特定应用场景的异常分析综述文章陆续发表出来. Chang 等人[23]首次对静态异常分析及其应用进行了综述, 指出静态异常分析的内容包括: 异常使用分析, 即异常处理结构的使用模式、使用静态分析对异常动态行为进行估计的分析等. 其中, 使用静态异常分析来估计异常的动态行为, 这类异常分析可以分为两类: 未捕获的异常分析和异常传播分析, 通过分析可以帮助程序员更合理、更有效地使用异常处理技术. 针对静态异常分析的应用包括: 对程序中的执行流(正常流和异常流)进行模拟, 将其分析结果应用在结构化测试、程序切片、控制流图的表示、编译器优化、开发可视化工具等方面. 毛澄映等人[24]对面向对象程序的异常分析与测试方法进行了评述, 首次按照异常的引发形式提出了显式异常与隐式异常的概念, 并根据使用的技术将异常分析的方法分为两大类: 基于程序结构的异常分析和基于形式推理的异常分析, 并从能够处理的异常类型、异常类型匹配的技术、异常流的表示形式、是否支持依赖性和切片、是否有支持的分析工具的角度对各种异常分析方法进行比较评述. 作为扩展, 他们也对异常分析工具进行了对比, 但文献[24]中只是针对面向对象语言中的异常分析及其应用进行了综述, 并未考虑其他特性的语言, 总结的异常分析方法依赖于具有一定特性的语言, 所提到的异常分析方法不具有良好的可扩展性, 并且没有考虑到库函数的特殊性, 只是将库函数中的异常归为隐式异常进行统一分析. 时隔 12 年, Chang 等人[25]在原有异常分析综述工作[23]的基础上, 从 4 个研究问题出发——异常分析的时机、异常分析的内容、异常分析的技术、异常分析的应用对异常分析进行了综述, 文中按照异常分析的时机将异常分析分为两大类: 静态异常分析和动态异常分析, 其中, 静态异常分析包括 3 类——异常的用途、程序中的未捕获异常、异常的控制流分析, 动态异常分析包括两类——测试异常和调试异常. 但是文献[25]中没有对可以获取部分源码的库函数进行异常分析总结, 且总结的异常分析方法不够全面, 例如, 基于函数摘要的异常分析方法[26]和基于符号执行的异常分析方法[18]没有包含其中, 没有归纳总结各类异常分析技术对库函数异常分析的适用性. 李忠和靳小龙[27]对于面向图的异常检测技术进行了研究综述, 然而他们讨论的异常是数据集中不同于其他数据的对象, 这与本文将要讨论的异常属于不同的范畴, 在此不对异常检测作过多的赘述.

尽管已有上述文献对软件供应链和异常分析进行了研究综述, 目前尚未有文章专门对软件供应链中库函

数的异常分析方法进行总结梳理与对比. 本文着重总结软件供应链中库函数相关的异常分析方法, 至于软件供应链的其他部分, 限于篇幅不再详述. 为此, 本文以软件供应链开源软件包中的库函数为研究对象, 对适用于库函数中异常处理结构的分析方法的基本思想以及重要步骤、优缺点进行系统的归纳、分类, 总结现有库函数异常分析在精度和效率优化方面所面临的挑战, 并给出初步解决思路, 这有别于现有的综述文章.

2 面向软件供应链的异常分析精度优化

本节从影响软件供应链中库函数异常分析精度的常见问题——确定引发异常的路径和不可达的处理程序出发, 针对库函数中异常分析精度优化方面的研究加以概述. 异常分析方法中, 以精度作为重要评价因素的分析方法包括基于抽象解释的异常分析[28−31]、基于集合约束的异常分析[32−40]、基于数据流的异常分析[41−56]、基于符号执行的异常分析[57−62]. 这些分析方法有助于对有高可靠性精度要求的库函数进行分析[63], 并且可以获得较大的回报. 异常分析是有效发挥静态分析优势的领域之一, 这些异常分析借助于一些程序分析方法对库函数中的异常处理结构进行分析, 以此获得库函数中由异常引发的异常控制流、所引发异常的传播路径、异常发生的位置以及库函数中未捕获的异常等信息.

2.1 库函数异常分析精度常见问题

下面以图 3 中的简单示例来说明库函数异常分析中与精度有关的常见问题——处理程序不可达, 使得程序中存在未捕获的异常. 为了简化描述, 将示例中的 *bar*(·)视为一个库函数, 假设类 *A* 和类 *B* 中都声明了方法 *bar*(·), 类 *A* 是 *B* 的父类, *bar*(·)可以视为是一个可以获取源码的库函数. 在类 *A* 的 *bar*(·)中抛出了一个 *IOException* 异常, 而在类 *B* 中的 *bar*(·)中没有抛出任何异常. 如果方法 *foo*(·)包含一个静态方法调用 *a.bar*(·), 那么 *foo*(·)必须为异常 *IOException* 定义一个处理程序或声明它抛出该异常. 但是, 如果在运行时引用 *a* 总是指向一个 *B* 类型的对象, 由于面向对象语言中的动态调用机制, 示例中的实际运行类型是 *B* 类型, *a.bar*(·)实际调用的是类 *B* 中重写的 *bar*(·)方法, 这样就不会抛出任何异常, 这将会使得处理程序中的 catch 子句不可达.

```
1   class A{
2     public:
3     void bar(·){
4       throw new IOException(·);
5   }
6   };
7
8   class B: public A{
9     public:
10    void bar(·){
11      cout<<"顺利执行!"<<endl;
12    }
13  };
14  void main (void){
15  A a=new B(·);
16  try{
17    a.bar(·);
18    }
19  catch (IOException e){
20    cout<<"IO 异常被捕获!"<<endl;
21  }
22  }
```

图 3 一个简单示例源程序

下面以上述图 3 所示类 throw-try-catch 异常处理机制为例, 对异常分析关注精度的分析优化方法的相关研究加以概述, 主要包括基于抽象解释的异常分析精度优化方法、基于集合约束的异常分析精度优化方法、基于数据流的异常分析精度优化方法和基于符号执行的异常分析精度优化方法. 这些方法对于库函数中异常处理结构的分析具有一定的适用性.

2.2 基于抽象解释的异常分析方法

抽象解释是一种对程序语义进行可靠抽象或近似的通用理论[26]. 抽象解释理论的一个重要思想是: 首先, 将标准语义转化为具体语义, 使得所分析的语义是有限的; 然后, 抽象化具体语义, 通过在抽象域上计算程序的抽象不动点来表达程序的抽象语义, 以此达到对具体域中的特性进行分析的目的.

基于抽象解释的异常分析[28−31]通常是将源程序转化为对应的中间表示, 对中间语言进行语义抽象, 将具体域的异常处理结构映射至抽象域的对应结构, 在抽象域中计算异常分析的迭代不动点, 即所有的异常相关语句(如 throw、try-catch、call 等)均已分析结束, 然后对异常相关抽象语义进行收集分析——语义的稀疏分析, 为程序的每个表达式计算一个值, 用来表示在该表达式中出现的运行时状态(包含未捕获异常的信息), 在不运行程序的前提下, 实现对程序的异常处理结构进行静态分析的目的, 最终可以获得每个处理表达式中引发的异常、全局的未处理异常.

下面阐述图 3 示例的基于抽象解释的异常分析过程[28,29].

(1) 首先, 需要将图 3 示例中的异常处理结构转化为对应的中间表示;
(2) 定义异常计算的标准语义, 为了使所获得的语义是有限的, 将其转化为具体语义;
(3) 对具体语义进行抽象, 使得可以在编译时对结果的解释进行计算, 包括对语义域和解释器函数的抽象;
(4) 在抽象语义中, 需要为源程序的每个分配点使用单个位置以消除不同环境的影响——定义函数时(在绑定表达式的内部)为函数名分配一个新位置, 应用一个函数时分配一个新的位置来保存函数的参数, 应用处理程序时保留异常值, 创建数据类型或异常值时分配一个新位置;
(5) 对引发异常和捕获异常的表达式的抽象语义进行计算, 将计算的结果映射至具体语义, 以实现对源程序中的异常处理结构进行分析的目的.

Yi 等人[28]基于抽象解释方法来检测标准元语言 SML 程序中引发但从未被处理的潜在运行时异常, 即: 该分析在程序执行前, 为输入程序的每个表达式计算一个值, 该值描述了在该表达式上发生的运行时状态, 通过程序状态来预测未捕获异常导致的异常终止, 以此来提高软件的安全性. 进一步地, Yi 等人为了在保证分析精度的前提下降低分析成本, 将上述文献[28]中的抽象解释加以拓展, 在文献[29]中引入了基于语义的未捕获异常稀疏分析, 文中对未捕获异常的稀疏分析适用于有源库函数中异常处理结构的分析.

Antoine 在文献[30]中将关系型抽象域应用于符合 IEEE 754 的浮点数的分析, 以便通过基于抽象解释的静态分析, 静态地检测潜在的浮点运行时异常, 如溢出或无效操作. 这种方法具有一定的通用性, 可以适应其他关系数值抽象域, 具有较好的扩展性, 同时具有良好的分析精度. 当对有源的库函数进行异常分析时, 可以按照文献[30]中的方法对库函数中的浮点数进行处理. Jaideep 等人[31]的方法有助于增强基于抽象解释的异常分析, 以获得由浮点计算引入的数值误差, 以便进一步地对程序中由浮点数引发的异常进行较高精度的分析.

2.3 基于集合约束的异常分析方法

集合约束[32]方法通过收集特定问题的相关集合约束信息, 并按照一定的约束求解规则对约束进行求解, 以达到对特定问题进行分析的目的.

基于集合约束的异常分析[32−37]方法包括两个阶段: 收集集合约束和求解约束. 通过收集异常相关集合约束信息, 并按照一定的约束求解规则对约束进行求解, 达到对异常处理结构进行分析的目的. 第 1 阶段通过推导规则构造约束, 以描述所分析程序的表达式之间的数据流; 第 2 个阶段是找到满足约束条件的值集, 常用的解决方法是使用一个从约束中的集合变量到可有限描述该类值集的表. 因为求解空间有限, 异常的名称是有限的, 可以通过迭代不动点进行计算求解, 实现对含有异常声明的语言进行异常分析. 这种方法可以较好地适用于软件供应链的场景.

现有的研究主要从表达式层面和方法/函数层面[33−37]进行分析, 图 3 所示的基于集合约束的异常分析示例的主要分析过程描述如下.

- 表达式层面的分析是对异常相关语句的每个表达式进行未捕获异常分析, 需要为每个表达式的对象类生成集合约束规则. 程序的每个表达式 e 都有一个约束: $X \supseteq X_e$, 其中, X_e 是表达式 e 的未捕获异常所属的异常类的集合变量, 主要需要生成引发异常表达式的集合约束规则、处理异常表达式的集合约束规则和方法调用表达式的集合约束规则. 例如图 3 示例中的语句 4、语句 16–语句 19 是异常相关的语句, 需要对这些语句的各个表达式建立异常约束规则;
- 方法/函数层面的分析是对异常相关方法进行整体的未捕获异常分析, 即对于一个方法 m 的所有子表达式有一个集合变量 X_m, 而不是对异常相关表达式中的所有子表达式进行分析, 其分析粒度较大, 是一种稀疏分析, 有效地保证了在相同精度下可以降低表达式异常分析的成本.

Yi 和 Ryu[32,33]设计并实现了基于集合约束的异常分析, 在调用图的基础上, 对程序中的异常流进行推导, 将推导的异常流用集合约束加以表示, 用于检测在 SML 程序中引发但从未处理过的潜在运行时异常. 这一分析将预测 SML 程序的突然终止.

Chang 等人[34]提出了一种基于集合约束的异常分析方法, 用来估计 Java 程序的异常传播信息. 为了将异常传播形式化, 描述了一个考虑了异常传播的操作语义, 基于这个操作语义估计异常传播, 有助于通过跟踪异常传播来将异常处理程序放在适当的位置. 在文献[34]的基础上, 该研究团队基于集合约束的分析框架[35,36], 为 Java 设计了一个过程间的未捕获异常分析. 该框架通过在程序点上设置约束来建模抛出的异常流, 可以独立于已声明的异常来估计未捕获的异常, 能够较为精确地检测 Java 程序中未捕获的异常.

进一步地, Yi 等人[37]扩展了 Java 语言以支持多线程异常处理, 并提出了一个静态线程间异常分析, 用于估计多线程 Java 程序中未捕获异常的传播. 未捕获的异常分析通常只通过其名称和结构来估计未捕获的异常, 因此它们不能提供关于抛出异常的传播路径的足够信息, 其分析精度有待于进一步的提高.

Chang 等人[38]设计并实现了一个异常传播分析, 它扩展了未捕获的异常分析[34], 以估计 Java 程序中抛出异常的可能传播路径, 并记录与异常相关的构造的标签来跟踪异常传播路径, 可以从分析信息构造一个异常传播图, 其中包括抛出异常的起点、处理程序和传播路径. 进一步地, Chang 等人[39]基于异常传播分析构建了一个可视化工具, 以促进理解程序的异常传播行为.

2.4 基于数据流的异常分析方法

数据流分析(data flow analysis, DFA)[41]是一种通过分析程序状态信息在控制流图中的传播来计算每个静态程序点(语句)在运行时可能会出现的状态, 在不同程序点上分析实时变量的定义-使用、到达定义、可用表达式的过程[42,43]. 按照分析过程中所关注的范围, 数据流分析[44]可分为过程内分析和过程间分析: 前者重点关注数据在同一函数内部的传递情况, 后者关注数据在若干函数间的传递情况.

现有的基于数据流的异常分析方法主要是针对异常处理结构建立过程间异常控制流图 IECFG[45]或过程间要素控制流图 IFCFG[46], 在图的基础上进行数据流分析. 其中,

- 基于过程间异常控制流图方法的主要思想[45]是: 通过类层次分析与指向分析相结合, 建立比较精确的调用图, 按照调用图的拓扑排序/逆拓扑排序顺序, 对每个过程建立并求解异常变量的“到达-定值”数据流方程, 静态地确定异常引发语句中异常变量的所有定值点; 按照异常控制流图获得异常的传播路径; 建立并求解异常变量的“定值-引用”数据流方程, 静态地确定异常捕获语句中异常变量的所有引用点集合, 从而可以确定程序中任何异常变量的“定值-引用”关系. 在分析过程中确定的定义-使用关系可以定义为异常捕获链, 即异常变量定义(异常引发点)和异常变量使用(异常捕获点)的路径;
- 基于过程间要素控制流图的异常分析方法在思想上与上述类似.

下面以图 3 所示为例, 分别阐述基于过程间异常控制流图和过程间要素控制流图的异常分析过程.

2.4.1 基于过程间异常控制流图的异常分析方法

- 过程间异常控制流图的定义

过程间异常控制流图(IECFG)是在传统控制流图的基础上, 针对异常处理结构添加异常相关的边和节点, 可以使用一个四元组来表示: $I_G=\langle N_s,N_e,N_{excepe},G_{inter}\rangle$, 其中, N_s、N_e、N_{excepe} 分别表示起始节点、正常退出和异

常退出节点, G_{inter} 是过程内异常控制流图的并集. 以此建立过程间异常控制流图. 为了准确地表示异常变量之间的关系, 需要把异常变量的定值和引用与异常控制流图中的节点联系起来.

图 3 所示源代码基于过程间异常控制流图的异常分析过程简述如下[45].

(1) 对于 IECFG 图中每一个包括异常变量的节点计算异常定义集; 为了计算到达 throw 节点的异常变量 v 的所有定值点集合, 需要建立数据流方程.

- $in[B]$: 到达基本块 B 入口之前的异常变量 v 的所有定值点集合;
- $out[B]$: 到达 B 出口之后(紧接着 B 出口之后的位置)的异常变量的所有定值点集合;
- $gen[B]$: B 中的定值并到达 B 出口之后的异常变量的所有定值点的集合;
- $kill[B]$: 基本块 B 外所定值的、在 B 中已被重新定值的异常变量的定值点集合.

$in[B]$和 $out[B]$的计算公式为 $out[B]=in[B]-kill[B]\cup gen[B]$, $in[B]=\cup out[P]$, 其中, P 是基本块 B 的所有前驱, 通过 IECFG 可以求得 $gen[B]$和 $kill[B]$集合, 进而按照公式可以求出 $in[B]$和 $out[B]$;

(2) 对于 IECFG 图中每一个包括异常变量的节点计算异常引用集; 按照步骤(1)中的相同方式, 结合过程间异常控制流图 IECFG, 可以计算到达 catch 变量的所有引用点集合;

(3) 对于一个异常变量和 throw 节点, 计算异常映射集. 按照 IECFG, 可以求出任一引发异常的传播路径, 沿着异常传播路径, 进而可以求得从 throw 语句可达的 catch 语句, 将异常类型与引发的异常类型进行匹配, 将该异常变量加入至异常映射集中;

(4) 计算异常变量在 throw 节点的“引用-定值”链;

(5) 计算异常变量在 catch 节点的“定值-引用”链.

其中, 步骤(4)和步骤(5)可以通过步骤(1)–步骤(3)计算得出.

2.4.2 基于过程间要素控制流图的异常分析方法

要素控制流图 FCFG 是一种比传统控制流图更加紧凑的控制流表示方式. 为了进一步提高异常分析的精度, 可以通过构建比传统的 CFG 基本块更大的 FCFG 基本块. 由于存在潜在异常抛出指令, 控制流可以从一个 FCFG 基本块的中间直接退出, 而无需像传统 CFG 那样——控制流只从一个基本块的末端退出, 即要素控制流图可以实现按需跳转. 在 FCFG 中, 可以对潜在异常抛出指令进行前向分析和后向分析, 因此, 这种双向分析方式可以提高异常分析的精度.

基于过程间要素控制流图的异常分析过程[46]描述如下.

(1) 构建过程内要素控制流图, 沿着异常控制流的执行路径, 进行前向和后向的数据流分析.

前向过程内数据流分析, 即到达-定值分析——首先, 为要素控制流图的每个基本块的非退出边计算 *gen* 集和 *kill* 集, 将每个基本块的 *gen* 集和 *kill* 集加以汇总; 使用汇总信息为每个基本块计算一个不动点; 将不动点的解传播到一个基本块内的指令中.

后向过程内数据流分析, 即活性变量分析. 首先, 为每个基本块中的活性变量计算 *gen* 和 *kill* 集, 汇总 *gen* 集和 *kill* 集; 为每个基本块退出计算不动点解; 在基本块中传播不动点的解.

(2) 在过程内要素控制流图的基础上建立过程间要素控制流图, 进行过程间分析.

利用过程内分析将调用点的信息传播到方法入口点, 并在方法出口点将异常相关的信息传播到调用点之后的指令.

- 构建过程间要素控制流图

在构建 IFCFG 时, 必须存在从潜在异常抛出指令(potential exception instruction, PEI)到捕获异常的处理程序的路径. 简单地, 从 PEI 到被调用方法的退出点建立边; 精确地, 在被调用方法中, 从 PEI 到处理程序建立边; 折中的方法是创建两个出口节点: 一个用于正常控制流, 另一个用于所有异常控制流.

Bundy 等人[15]指出, 自动化静态分析很适用于识别大型 Ada 系统异常处理代码中的缺陷. 文献[15]中描述了一个用于分析 Ada 代码中的异常处理的工具, 并讨论了使用该工具来检测违反特定于应用程序的设计和编码准则的情况. 但是该文没有进行数据流分析, 因此异常分析的结果不够精确. 这也反映出数据流分析有

助于提高异常分析的精度.

Chatterjee 和 Ryder 在文献[47]中指出: 由于异常和动态调度的存在, 使得面向对象编写的库中寻找定义-使用关系是非常困难的. 该文首次针对面向对象库提出了一种流敏感和上下文敏感的数据流异常分析方法, 该方法可以处理不完整的面向对象程序/库, 同时考虑到显式抛出异常的处理.

Robillard 和 Murphy[48]设计并实现了一个名为 Jex 的工具, 该工具基于异常流模型对 Java 执行一个未捕获的异常分析. 异常流模型侧重于描述基于数据流分析框架程序中抛出的异常的可能流向.

Fu 和 Ryder[49]提出了一个基于数据流分析框架的未捕获异常分析, 该框架将异常流标识为异常的"定义-使用", 它对应于抛出和捕获异常, 该分析信息用于错误恢复代码的白盒覆盖测试, 提出了一个程序可视化工具 ExTest, 它非常精确地显示了由某些类型的操作触发的异常以及它们的处理程序, 对于每个处理程序, 将如何触发该异常操作的所有路径进行了展示. Fu 和 Ryder[50]还通过扩展未捕获的异常分析[49], 提出了一种基于异常链分析的异常传播分析方法, 它可计算与语义相关的异常链, 即异常抛出语句到异常捕获语句之间的定义-使用关系, 从而报告整个异常传播路径, 而不是像文献[49]那样只报告它们的离散段. 这种分析方法更加精确, 主要是通过指向分析构建比较精确的调用图, 在调用图的基础上进行异常流分析和数据可达性分析, 从而获取真正可行的异常捕获链.

Weimer 和 Necula[50]提出了一种数据流分析, 用于查找那些由于未能释放资源以及沿着路径进行正确清理时产生的错误.

Buse 和 Weimer[51]提出了一个基于过程间上下文敏感的全自动的工具, 它可以静态地推断和描述 Java 程序中导致异常的条件, 且该工具可以自动生成异常相关文档, 并帮助开发人员预判可能发生异常的条件.

Hovemeyer 和 Pugh[52]开发了一个 bug 发现工具 FindBugs, 它可以自动发现程序源代码中的编程错误. 他们关注无效异常和其他异常, FindBugs 基于数据流分析来报告潜在的空指针异常, 该分析跟踪异常路径上潜在的空指针解引用缺陷.

Wu 等人[53]提出了一种将前向流敏感分析和后向路径可达性分析相结合的静态分析方法, 以检测 Java 程序中不正确异常处理所导致的错误, 确定每个不安全的警告是否为假阳性, 以此提高异常分析的精度.

姜淑娟等人[54]以分析 C++异常传播机制对数据流分析的影响为基础, 提出了一种基于异常传播机制的数据流分析方法. 该方法考虑了 C++过程间异常传播, 通过构建过程间异常控制流图, 对 C++中的异常处理结构进行精确分析.

Liblit 等人[55]指出了一种基于数据流分析的加权推导系统来解决错误传播的问题, 使用错误传播信息来识别不同的错误处理缺陷.

Prabhu 等人[56]对异常控制流图进行扩展, 提出了一种基于有符号类型集域的对 C++程序进行过程间异常分析的方法, 将过程内的异常控制流图扩展至过程间, 并将带有异常结构的 C++程序通过降低代码的方式转化为无异常的 C 程序, 对转化成的无异常程序进行数据流分析, 实现了对 C++程序的精确异常分析.

2.5 基于符号执行的异常分析方法

符号执行[57,58]是一种相对精确的程序分析技术, 以程序的完整执行状态为模型, 采用符号化输入代替实际输入, 以模拟执行被分析的程序. 在分支语句处, 对特定问题有影响的变量进行符号化表示, 对程序中的各个分支进行符号化分析, 产生符号化的执行路径. 在分析过程中, 收集特定问题需要的符号化约束条件, 使用约束求解器对约束条件进行求解, 得到的解可以标识该执行路径是否可行. 由于异常处理结构相关语句在有源库函数中较为稀疏, 所收集的约束条件较少, 因此, 在符号执行路径条件数量可承受的范围内进行路径约束条件的求解, 可以最大化地发挥符号执行的优势.

传统的符号执行是静态的, 由于实际程序的符号约束有时并不能被约束求解器求解, 当遇到源代码不可获得的函数调用时, 例如第三方库, 为了实现这类库函数的异常分析, 动态符号执行[18]可以有效解决这类问题, 它在传统的符号执行中利用符号的实例化提高了可行性[59], 即动态地运行被测程序, 实际运行的同时收集运行路径上的路径条件, 然后对路径条件取反得到新的路径条件, 通过求解新的路径条件得到新的程序输

入来再次运行被分析程序, 从而探索与之前不同的程序路径. 当动态符号执行遇到某些不可求解的输入时, 就使用它的运行时实际值来化简约束, 再进行求解, 从而使测试尽可能地覆盖更多的分支和路径.

下面通过图 3 所示示例来介绍基于符号执行的异常分析[60]的主要过程.

(1) 首先, 确定程序中与异常相关的待符号化变量, 一般包括库函数的形参、引发异常的分支语句中的变量和全局变量, 对这些变量进行符号化;

(2) 收集可能引发异常的路径条件, 将路径条件转化为符号化的表达式;

(3) 使用符号执行约束求解器对收集的路径条件进行求解, 获取可行的路径, 确定程序中真正可行的异常捕获链, 即从异常引发到异常捕获的可行路径.

Thomas 等人[31]指出符号执行工具在软件中能够发现细微的错误, 这方面的表现非常成功, 他们提出了一种基于符号执行的方法来检查数值型程序的准确性, 研究浮点型计算与实际值的“理想”计算偏离问题, 他们在文献[31]中提到的方法可以适用于浮点异常检测的相关研究工作中, 且文中明确说明: 使用所开发的工具可以提高基于抽象解释分析方法的精度, 以减小由浮点计算引入的数值误差.

Tomb 等人[61]使用符号执行来查找可能发生异常的约束, 然后对这些约束条件进行求解, 生成可能会引发异常的测试输入.

Belyaev 等人[18]提出了一种基于符号执行的异常分析方法来管理在执行 C#程序时的异常, 该方法能够处理 C#语言的最新特性——捕获过滤器. 他们所提方法非常精确, 是一种路径敏感且字段和对象敏感的方法.

Marcelo 等人[62]提出了一种基于符号执行的测试数据生成环境中的异常情况分析方法, 他们在文献[62]中指出: 激活特定异常所需约束可能不能在代码中直接声明, 在这种情况下, 必须从异常处理机制中推断出隐式约束. 文中所使用的工具 CP4SE 可根据字节码符号化地执行正在测试的程序, 并为每条执行路径生成异常相关路径约束.

3 面向软件供应链的异常分析效率优化

本节对关注软件供应链中库函数异常分析效率的研究工作进行概述, 主要对有效提高库函数异常分析效率的两大类方法进行总结归纳, 包括基于类型系统的异常分析方法[64−82]和基于函数摘要的异常分析方法. 其中, 基于类型系统的异常分析方法按照特定的异常相关类型规则来预判库函数中可能存在的异常, 基于函数摘要的异常分析方法对于有源、无源和半有源的库函数都具有较好的适用性.

3.1 基于类型系统的异常分析方法

类型[64]于 20 世纪 50 年代被 FORTRAN 语言引入, 其相关的理论和应用[65,66]已经发展得非常成熟. 现在, 类型系统已成为各大编程语言的核心基础. 类型系统是一门编程语言不可或缺的部分, 它的优势有以下几个方面.

(1) 排查错误. 很多编程语言都会在编译期或运行期进行类型检查, 以排查违规行为, 保证程序的正确执行. 如果程序中有类型不一致的情况, 或有未定义的行为发生, 则可能导致错误产生. 尤其是对于静态语言来说, 能够在编译期排查出错误是一个很大的优势, 这样可以及早地处理问题, 而不必等到运行后类型系统崩溃再解决问题;

(2) 类型安全. 类型安全的语言可以避免类型间的无效计算, 例如可以避免不符合算术运算规则的计算. 类型安全的语言还可以保证内存安全, 避免诸如空指针、悬垂指针和缓存区溢出等导致的内存安全问题. 类型安全的语言可以避免语义上的逻辑错误.

基于类型系统的异常分析方法主要是在已有类型系统的基础上添加异常处理结构相关语句的类型规则, 按照类型规则之间的关系, 采用归约理论推导出所需要的异常类型信息(不仅仅是类型相关的异常), 最后通过某种转换机制, 确定是哪一种异常. 例如: 在对各个子过程进行异常分析时, 可能会同时存在多个异常, 在这种情况下, 可以借助类型系统的归约思想, 首先对多个异常进行类型归约, 这样可以有效避免将过程内分析得到的多个同类型异常同时传送至过程间, 降低过程间分析的时间开销, 从而保证在不损耗分析精度的同

时, 提高异常分析的效率.

以图 3 所示类 throw-try-catch 异常处理结构为例, 与异常相关的类型系统规则[67]可以如下设定.

(1) 异常构造

$$\frac{\vdash \varphi' :: EXN(\{C\}) \ \vdash \varphi :: EXN(\phi)}{E \vdash C : exn[C : pre; \varphi'] / \varphi}$$

$$\frac{\tau \leqslant TypeArg(D) \ E \vdash a : \tau / \varphi \ \vdash \varphi' :: EXN(\{D\}) \ \vdash \varphi :: EXN(\phi)}{E \vdash D(a) : exn[D(\tau); \varphi'] / \varphi};$$

(2) 异常处理

$$\frac{E \vdash a_1 : \tau / \varphi' \ E \oplus \{x : exn[\varphi']\} \ \vdash a_2 : \tau / \varphi}{E \vdash \text{try } a_1 \text{ with } x \to a_2 : \tau / \varphi};$$

(3) Match 异常类型匹配

$$\frac{E \vdash a_1 : \tau_1 / \varphi \ \vdash p : \tau_1 \Rightarrow E' \ \vdash \tau_1 - P \to \tau_2 \ E \oplus E' \ \vdash A_2 : \tau / \varphi \ E \oplus \{x : \tau_2\} \ \vdash a_3 : \tau / \varphi}{E \vdash \text{match } a_1 \text{ with } \ p \to a_2 \mid x \to a_3 : \tau / \varphi}.$$

Leroy 和 Pessaux[67]设计并实现了一种基于类型和效果系统的 OCaml 未捕获异常分析, 并在真实的 OCaml 程序中显示了良好的性能. Allen 等人[68]指出: 静态地分析库需要对所有事先未知的可能应用程序进行新的抽象, 设置特定问题对应的类型系统规则有助于进一步的抽象. 库为应用程序提供了一个公共接口, 并且库所显示的行为由应用程序控制. 文献[68]中展示的方法能够为具有 JNI 本地方法的库和应用程序未知的库抽象出堆这一数据结构, 其中, 抽象未知应用程序行为的一种解决方案是将类型分析与指向分析相结合.

Guzman 和 Suarez[69]提出了一个 SML 类型系统的扩展, 通过它, 可以静态地估计程序中产生的所有未捕获的异常, 该类型的系统可以处理关于异常的多态信息. Danie 和 Todd[70]提出了一个通用的类型和效果系统. Giord 和 Lucassen 等人[71]在特定上下文中对一种类似于 Scheme 的语言中命令式表达式进行检测时, 首次利用了用表达式的操作属性来装饰箭头的想法(即在箭头上添加注释信息), 并在文献[72]中使用类型中包含效果的集合关系来推导类型的子集关系, 以此来描述类型效果推理规则. Leroy 和 Weis[73]研究了注释类型, 他们以如何多态地类型化 SML 的引用为研究重点, 所提出的系统能够非常成功地从安全的引用中识别危险的引用. Benton 和 Buchlovsky[74]提出了一个多态类型和效果系统, 该系统跟踪像 SML 这样的严格函数式语言可能抛出的异常和可能的非终止情况, 并能够很容易地将定理证明类型分析和转换的一般关系扩展到异常分析的情况下.

Glynn 等人[75]首次为 Haskell 提出了未捕获的异常分析, 他们设计了一个基于类型的推理系统来检测未捕获的异常, 并在 Haskell 的 GHC 编译器中得以实现. Koot 和 Hage[76]为 Haskell 设计了一种基于类型的异常分析, 该分析可以检测部分函数的有害源头, 并证明其对于不精确的异常语义是正确的, 同时分析了跟踪异常流和普通数据流以及它们之间的依赖关系.

Yi 等人[77]提出了一个基于类型的静态分析方法, 它首先从来自 SML/NJ 编译器的类型信息中估计输入程序的控制流, 即借助类型推断阶段的信息来完成一种粗糙的闭包分析, 可以检测在 SML 程序中引发的潜在运行时异常, 并可减少异常分析的时间开销. 为了保证在不损失分析精度的前提下提高分析速度, Fähndrich 和 Aiken 在文献[40]中提出将基于类型系统的等式约束(速度快)与基于集合的包含约束(精度高)相结合, 可以达到这一目的. 此外, Laud 等人[78]提出: 在命令式语言中, 基于类型系统的分析方法与数据流分析方法是等价的, 可以保证在不损失精度的前提下提高分析的效率.

Sylvain[79,80]提出了一个系统 F 的扩展类型系统, 用来检测按名称调用的异常, 所设计的类型系统具有两种语法结构: 一种联合类型用于执行可能在顶层引发异常的程序, 另一种损坏类型用于可能在任何求值上下文中引发异常的程序(不一定在顶层).

Maria 等人[81]设计了一个类型系统来检查可能接收作为外部输入传递的错误格式值的方法. Yi 等人[82]提出了一种支持多态模式的类型系统及其主要类型推理算法, 该类型系统支持开放代码, 对引用可以进行不受限制的操作. Yi 等人[83]提出了一个带异常的多阶段语言的类型效果系统, 所提出的类型和效果系统用来检查

能否在多阶段编程中安全地合成带异常的复杂控制.

3.2 基于函数摘要的异常分析方法

函数摘要是一种使用函数的摘要信息来表示函数的方式, 张健等人在文献[26]中指出: 在程序分析之前对常用代码库生成摘要, 可以加快使用这些代码库的客户端的分析速度. 如果程序中存在大量的函数调用, 那么在进行过程间异常分析时, 会带来巨大的时间空间开销. 在遇到函数调用时, 对被调用函数进行分析提取其函数摘要, 然后在调用点应用该函数摘要, 以后每次遇到关于该函数的调用, 都可以使用函数摘要代替其函数. 由于只是在提取函数摘要时才会对被调用函数进行分析, 因此函数摘要不会重复分析函数, 从而能够提高过程间分析的效率.

基于函数摘要的过程间异常分析[26]会在过程内异常分析的基础上收集被调用函数的异常信息, 在遇到关于该函数的函数调用时, 就不需要对该函数进行内联展开, 而是直接使用对被调用函数进行过程内异常分析得到的摘要信息, 再进行过程间的异常分析, 这是一种自顶向下的分析方式. 现有程序使用了大量的基础库, 这些库函数比较通用且函数行为很固定, 可以预先对这些库代码进行异常分析, 收集其函数摘要信息, 从而加速整个分析过程. 对于无法获取源码的库函数, 例如第三方库等, 可以采用人工编写摘要近似代码的行为. 在人工编写函数摘要的基础上, 进行过程间的异常分析.

王留帅在文献[84]中指出: 函数摘要的设计是过程间异常分析精确与否的关键, 函数摘要能否真正代表该函数在函数调用处对异常上下文的影响, 是函数摘要设计的首要准则. 其中, 异常上下文是指异常从引发到捕获所在的环境, 即异常引发的上下文是异常的捕获, 异常捕获的上下文则是异常捕获后进行的后续操作. 目前较为通用的函数摘要是符号化函数摘要, 符号化函数摘要的设计思想是: 将函数摘要中依赖于异常上下文的信息符号化, 在函数调用点应用函数摘要时, 使用上下文环境信息对函数摘要进行实例化, 从而为函数创建一份符号化的函数摘要, 就能实现上下文敏感的分析.

基于函数摘要的异常分析主要过程[84,85]概述如下.

(1) 函数摘要的创建

首先, 按照 AST 和异常控制流图 ECFG 结合类层次分析和指向分析建立精确的函数调用图, 对函数调用图进行拓扑序(执行方向是自顶向下)或逆拓扑序(执行方向是自底向上), 确定各个过程的分析顺序; 以自底向上的分析为例, 按照逆拓扑序选取一个待分析的过程/库函数, 对该过程/库函数进行过程内的异常分析, 这一步骤可以根据一些常用的过程内异常分析方法来进行. 所有的库函数过程内异常分析结束后, 将引发异常的条件(从所有入边的异常退出块中收集异常值, 将异常引发条件和所引发异常的相关信息存储在函数摘要中)或过程内分析符号化的程序状态作为函数摘要信息. 该过程可以结合符号执行技术精确化地加以分析, 以获得符号化的函数摘要信息.

(2) 函数摘要的应用/实例化

如果在过程内分析过程中遇到了函数调用语句, 首先检查被调用函数是否存在函数摘要信息: 如果存在, 则直接获取其函数摘要信息, 然后在函数调用点对函数摘要进行实例化, 并应用到上下文环境中(例如, 将被调用者函数摘要中的异常 valueId 转换为调用者上下文); 如果还没有为该函数计算过函数摘要, 则将该函数作为一个单独的部分, 通过过程内异常分析方法进行过程内分析, 在得到函数摘要信息后, 在函数调用点对该摘要信息进行实例化, 并应用到上下文环境中.

赵云山[86]实现了基于符号化函数摘要的静态分析方法及系统.

董玉坤等人[87]针对空指针解引用缺陷提出了一种基于区域内存模型的函数摘要框架, 该方法可以对不包含指针的程序进行稀疏异常分析, 该函数摘要只编码了内存相关的摘要信息, 是一种路径不敏感的分析, 其分析精度有待进一步提升.

王留帅在文献[84]中使用基于函数摘要和符号执行相结合的方法对程序进行路径敏感的过程间分析, 函数摘要是一种路径敏感的符号化函数摘要, 文献[84]中所提方法的思想对于异常分析具有很大的借鉴意义. 其中, Belyaev 和 Ignatyev[18]采用了这种思想, 以此来分析 C#中带有新特性的异常处理结构, 使用符号执行来

保证异常分析的精度, 使用函数摘要来提高异常分析的效率. 这种分析方法同样适用于对外部库函数和虚函数的异常分析, 但是现有实现的分析比较保守, 还需要进一步提高分析精度.

4 总结与展望

本文从影响软件供应链安全的库函数(包括开源和闭源的库函数)出发, 总结概述了可以适用于库函数的异常分析方法, 在程序运行之前, 对库函数可能引发的异常/错误进行预判和修复, 可以有效防止程序中可能出现的漏洞发生. 本节对关注软件供应链中库函数异常分析效率和精度的研究进行了归类整理, 并对面向软件供应链中库函数异常分析可能面临的挑战提出了初步的解决思路, 并对未来的发展方向加以展望, 以此提供一种面向软件供应链安全的有效且精确的防范措施, 对保证软件供应链安全具有一定的借鉴意义.

4.1 面向软件供应链的异常分析方法分类比较

为了更加形象、直观, 本文进一步根据所关注的角度, 对方法的主要步骤以及各种异常分析方法的优势与不足进行了分类归纳, 简洁描述见表 1.

表 1 各种异常分析方法汇总表

关注角度	分析方法	方法概述	方法的优势	方法的不足
精度	基于抽象解释的异常分析	将源程序转化为对应的中间表示, 对中间语言进行语义抽象, 将具体域的异常处理结构映射至抽象域的对应结构	考虑了接收对象的类型信息, 将这些类型信息映射至抽象域中, 具有较好的分析精度	仅限于类型正确且操作语义不变的程序, 不考虑数据类型和异常声明, 其抽象状态表示与计算的时空开销较大, 分析速度不够快, 无法进行交互使用
	基于集合约束的异常分析	收集集合约束和求解约束. 通过收集异常相关集合约束信息, 并按照一定的约束求解规则对约束进行求解	可以相当准确地模拟程序内的值流方向, 使得对库函数的异常分析结果较为精确	软件规模的增大会使得在求解约束时的成本增加, 约束求解较为困难, 可扩展性较差
	基于数据流的异常分析	针对异常处理结构建立过程间异常控制流图 IECFG 或过程间要素控制流图 IFCFG, 在图的基础上进行数据流分析	无需依赖于任何形式化软件规范, 可以将异常相关的信息在图上直观、精确地表现出来, 较为精确地分析库函数的异常处理结构	构建过程间异常控制流图的时空开销非常大, 而且图一旦构建将很难进行自动更新, 在一些需要动态更新的场景, 这种方法不具有良好的可扩展性
	基于符号执行的异常分析	采用符号化输入代替实际输入, 在分支语句处对特定问题有影响的变量进行符号化表示, 分析过程中收集特定问题需要的符号化约束条件, 进行约束求解	适用于无法获取源代码的库函数的异常分析, 可以借助动态符号执行提高分析精度	在进行大规模程序异常分析时, 由于所收集的路径约束条件较多, 在进行约束求解时效率有待提升
效率	基于类型系统的异常分析	在已有类型系统的基础上添加异常处理结构相关语句的类型规则, 采用规约的理论, 推导出供应链异常的类型信息	可以在不支持类型系统的语言上, 设定符合特定要求的类型规则	对于支持类型系统的新型语言中的新特性需要添加额外的类型规则, 类型系统较为繁琐, 不易理解
	基于函数摘要的异常分析	在遇到函数调用时, 对被调用函数进行分析, 提取其函数摘要, 然后在调用点应用该函数摘要	比较适合对无源库函数的异常分析, 无需重复分析函数, 从而提高过程间分析的效率	函数摘要是离散化的一些具体值或符号化的值, 无法将结果保存为连续的值, 不能进行有效的扩展

从抽象解释技术的局限性来看, 由于需要建立具体域和抽象域之间的映射关系, 其抽象状态表示与计算的时空开销较大, 分析速度不够快, 无法进行交互使用. 但这种分析方式由于考虑了接收对象的类型信息, 因此, 可将这些类型信息映射至抽象域中. 这种分析方法仅限于类型正确且操作语义不变的程序, 不考虑数据类型和异常声明, 因此, 基于抽象解释的异常分析方法为了能够更好地适用于大型的库函数异常分析, 需要进行全局的稀疏分析, 以便取得更好的精度.

基于集合约束的异常分析可以尽可能地保证分析精度, 并最大程度地在保持精度的同时降低分析成本.

正如文献[40]所指出的: 基于集合约束的异常分析可以相当准确地模拟程序内的值流方向, 使得对库函

数的异常分析结果较为精确. 随着软件规模的扩大, 该方法在求解约束时的成本增加, 约束求解较为困难, 可扩展性较差, 需要简化集合约束的表示, 进行异常相关的稀疏分析以降低时空开销.

基于数据流的异常分析方法无需依赖于任何形式化软件规范, 有助于开发自动化的异常分析工具, 是在过程间异常控制流图或要素控制流图的基础上进行异常处理结构的分析. 为了将异常引起的控制流传递至过程间, 需要在传统控制流图的基础上添加额外的节点或边. 这种基于图的方式可以将异常相关的信息在图上直观、精确地表现出来, 可以准确地标识由引发异常导致的异常控制流, 包括异常的引发点和异常的处理点以及异常处理后的程序的准确流向. 这种基于数据流的异常分析方法可以较为精确地分析库函数的异常处理结构, 可与其他异常分析方法相结合, 进一步提高异常分析的精度. 但是, 这一分析方法随着软件系统复杂性的提高, 将会使得构建过程间异常控制流图的时空开销非常大, 而且图一旦构建将很难进行自动更新, 故在一些需要动态更新的场景下, 这种方法不具有良好的可扩展性.

基于符号执行的异常分析方法可以有效提高异常分析的精度, 与其他分析技术相结合可以有助于在异常分析精度和效率方面取得最大化平衡, 尤其是对于无源库函数的异常分析, 可以借助于动态符号执行获取一个较好的精度. 但现有的基于符号执行的异常分析方法也存在一些不足之处, 例如: 面向多语言编写的库函数的异常分析场景, 符号执行需要适应多语言的不同特性, 以最大化地发挥符号执行的优势. 在进行由库函数组成的大规模程序的异常分析时, 由于所收集的路径约束条件较多, 在进行约束求解时, 效率有待提升.

对于软件供应链中库函数的异常分析, 可以在不支持类型系统的语言上设定符合特定要求的类型规则, 采用类型约束的思想, 解决传统的异常分析解决不了的问题. 基于类型系统的异常分析可以与其他静态分析方法相结合, 进一步提高异常分析的效率.

正如文献[88]所指出的: 基于函数摘要的分析方法比较适合对库函数的异常分析, 尤其是对于无源库函数具有较好的适用性. 由于对无源库函数分析所获得的函数摘要采用人工编写的方式, 虽然这种摘要可以模拟被调用库函数中异常引起的数据流路径, 但没有考虑用户自定义的库函数对被调用库函数的影响, 分析的精度有待进一步提高. 从整体来看, 虽然这种分析方法可以避免对被调用过程的重复分析, 但是这些函数摘要是离散化的一些具体值或符号化的值, 无法将结果保存为连续的值(甚至函数), 不能进行有效的扩展. 函数式编程中的高阶函数特性能够弥补这一点, 可以将摘要设计为高阶函数式的摘要, 即将过程内的分析结果保存为一个函数, 并且这个函数能够作为函数参数进行传递, 从而可以进一步提高库函数异常分析的效率, 这种分析方法在软件供应链场景下具有较好的适用性.

4.2 面向软件供应链中库函数异常分析的挑战和解决思路

通过上述对各种异常分析方法的优缺点进行归纳总结, 面向软件供应链中库函数异常分的研究仍然面临以下挑战.

(1) 由面向对象语言中的一些特性(例如动态调度/虚函数调用/多态)引起的分析精度损失问题. 正如图 3 所示源代码, 动态调度或虚函数调用[89,90]将引起异常分析的精度损失问题. 虽然已有上述异常分析方法, 但在大规模程序中, 由于动态调度, 使得调用的目标取决于接收对象的运行时类型, 而虚函数调用在运行时才能知道真正调用的虚函数, 所以无法完全静态地精确获知具体的被调用函数, 这使得构建健全和精确的调用图通常是困难和昂贵的. 调用图确定了程序中的调用顺序, 因此, 构建精确的调用图对于异常分析的精确性尤为重要, 现有异常分析方法的精度有待于进一步的提升;

(2) 无源库函数异常分析可拓展性问题. 根据第 3.2 节对基于函数摘要的异常分析方法的概述, 虽然符号化的函数摘要方法具有一定的可扩展性, 但是由于符号化函数摘要算法[84]对于无源库函数并未进行有效的建模, 而工程级源码中存在大量的库函数调用, 并且大部分库函数的定义无法获得, 因而极大地限制了跨过程分析的能力, 这种方法有待于进一步的改进;

(3) 新型语言中的特性, 使库函数异常分析变得困难. 新型语言(例如 Rust 和 Go)[91−94]编写的程序越来越普遍, 但是对于 Rust 和 Go 中的错误处理结构进行分析的研究较为稀少. 以 Rust 库函数中的 unsafe 代码为例, 这些 unsafe 代码破坏了整个 Rust 程序的安全性. 因此, 彻底地分析 Rust 程序和库

函数以排除错误是至关重要的. 尽管目前已有很多研究型项目, 比如形式化类型系统和操作语义, 通过建立模型以用于验证程序的正确性, 但是它们的可扩展性还不够强, 难以覆盖到整个 Rust 生态系统. 当前, 对于库函数的分析依赖于程序员手动构造测试用例, 这是一个低效的过程. 而 Rust 编译器具有严格的安全规则检查, 这给使用自动化方法构造通过编译的测试用例带来了一些挑战. 因此, 研究面向库 API 生成有效的 API 调用序列、合成类型良好的 Rust 程序、在错误分析过程中考虑 Rust 的所有权类型系统和多态性, 这些是亟需解决的问题;

(4) 库函数调用可能产生回调函数问题, 使库函数异常分析函数摘要的建立变得困难. 对于简单的库函数, 现有的函数摘要生成方法可以使用建模的方式产生摘要, 但是对单独的库函数进行分析所获得的摘要信息可能是不全的. 此外, 当库函数通过函数指针调用函数时, 不会对应用程序代码进行分析, 这将会引发回调函数问题[88,95−97]. 由于无法确定实际回调的函数, 无法获知库函数的数据流正确走向, 这将使得函数摘要的建立变得较为困难, 降低了函数摘要这种方法的有效性;

(5) 对并发场景中库函数调用产生的新式异常分析较少. 以并发程序中对库函数的调用为例, 由于不同应用程序对库函数的同时调用, 可能会引发数据竞争相关的异常, 称之为数据竞争异常, 这类异常在有关文献[98−100]中已被明确提出. 如果该类异常没有得到有效的处理, 将会产生一些安全隐患, 对软件供应链的安全产生威胁, 所以文献中提到的数据竞争检测方法对数据竞争异常的分析具有一定的借鉴意义, 从而减少了该类异常的不当处理对软件供应链安全产生的影响.

面对以上主要挑战, 函数式编程的相关特性和类型系统的特有优势可以为库函数异常分析提供一种可行的解决思路. 函数式编程中的高阶函数特性[101], 有助于提升基于函数摘要供应链异常分析的效率. 高阶函数特性是指函数可作为其他函数的返回值又可作为其他函数的参数, 可以根据需要生成高阶函数形式的摘要, 与具体值的函数摘要形成了显著的差异.

(1) 首先, 对所有可以获取到源码的库函数, 沿用基于函数摘要的异常分析方法对库函数进行分析, 生成高阶函数式的库函数摘要. 在过程间异常分析时, 按照形实参之间的对应关系, 将高阶函数式的库函数摘要在函数调用点进行实例化. 进一步地, 借助于延续传递风格(CPS)[102−105], 可将高阶函数式的函数摘要串成一个链式结构, 以提高异常分析的效率, 解决对库函数进行调用所导致的回调函数问题和一些安全性问题, 达到在按需调用库函数的同时仍能保证安全的目的;

(2) 对于获取不到源码的库函数(例如第三方库), 如果可以获取到输入参数, 例如接口信息, 也可以借助高阶函数进行分析. 这是因为可以根据形实参的对应关系进行异常分析相关信息的传递, 不需要获取到完整的函数体, 也可以通过库函数的接口和库函数提供的分析工具获得异常信息, 例如 ExceptionId 等, 这可以与直接分析源码实现等效分析, 进一步进行后续分析; 借助情形(1)和类型系统的特性, 设置一个异常有关的数据结构, 例如 Normal|Exception, 若异常被捕获, 则将 ExceptionId 置为 0; 若未被捕获, 则将 ExceptionId 置为 1.

4.3 展 望

现有的软件项目越来越依靠于第三方和开源组件, 该类组件往往由个人创建和维护, 由于这类组件与开发重要软件的组织无特定的雇佣关系, 所以第三方不一定使用与软件开发组织相同的安全策略, 因此存在着一定的安全风险. 攻击者可以通过多种方式危害软件供应链安全, 其中包括利用第三方组件中存在的错误或漏洞来破坏第三方开发环境, 并向其中注入恶意代码. 由于软件供应链的安全性与软件系统的健壮性密切相关, 因此, 对软件供应链中的重要组成部分——库函数(包括第三方库)中的错误/异常进行分析, 有助于提高软件系统的健壮性, 增强软件供应链的安全. 本文中总结的异常分析方法虽然对软件供应链中的库函数具有一定的适用性, 但是对于特定场景下的适用性需要进行实验验证, 其分析精度和效率可以进一步加以提高; 并且本文着重总结软件供应链中库函数相关的异常分析方法, 至于软件供应链其他部分的异常分析方法, 也将是我们下一步的研究工作. 例如, 函数式编程中的高阶函数特性有助于优化基于函数摘要的异常分析方法, 在现有静态分析的基础上添加类型规则, 借助符号执行思想进行动态扩展, 使其更加精确和高效.

References:

[1] National Natural Science Foundation of China. Development Strategy of Software Science and Engineering Discipline. Beijing: Chinese Academy of Sciences, 2021 (in Chinese).

[2] Bradley K, Godfrey M. A study on the effects of exception usage in open-source C++ systems. In: Proc. of the IEEE 19th Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM). 2019. [doi: 10.1109/SCAM.2019.00010]

[3] Su Q, Zhao R. Security risk analysis and development suggestion of open source software supply chain. Journal of Information Security Research, 2022, 8(8): 831−835 (in Chinese with English abstract).

[4] Sena D, Coelho R, Kulesza U, Bonifácio R. Understanding the exception handling strategies of Java libraries: An empirical study. In: Proc. of the 13th IEEE/ACM Working Conf. on Mining Software Repositories (MSR). Austin, 2016. 212−222. [doi: 10.1145/2901739.2901757]

[5] Miao ST. The research of security inopen source software supply chain. Secrecy Science and Technology, 2020(7): 37−42 (in Chinese with English abstract).

[6] He XX, Zhang YQ, Liu QX. Survey of software supply chain security. Journal of Cyber Security, 2020, 5(1): 57−73 (in Chinese with English abstract).

[7] Xiao GD, Ye RG, Jiao CP. Analysis of security problems in software supply chain in China and research on countermeasures. Information Technology and Standardization, 2020(6): 49−52 (in Chinese with English abstract).

[8] Dong GW. Analysis and countermeasures on current situation of software supply chain security. China Information Security, 2021(10): 34−37 (in Chinese with English abstract).

[9] Lang J, Stewart DB. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. ACM Trans. on Programming Languages and Systems, 1998, 20(2): 274−301. [doi: 10.1145/276393.276395]

[10] GoodEnough JB. Exception handling: Issues and a proposed notation. Communications of the ACM, 1975(12): 683−696. [doi: 10.1145/361227.361230]

[11] GoodEnough JB. Structured exception handling. In: Proc. of the Conf. Record of the 2nd Annual ACM Symp. on Principles of Programming Languages. 1975. 204−224.

[12] Gehani NH. Exceptional C or C with exceptions. Software: Practice and Experience, 1992, 22(10): 827−848. [doi: 10.1002/spe.4380221003]

[13] Cui Q, Gannon J. Data-oriented exception handling [Ph.D. Thesis]. University of Maryland, 1989.

[14] Sinha S, Harrold MJ. Analysis and testing of programs with exception handling constructs. IEEE Trans. on Software Engineering, 2000, 26(9): 849−871. [doi: 10.1109/32.877846]

[15] Schaefer CF, Bundy GN. Static analysis of exception handling in Ada. Software Practice and Experience, 1993, 23(10): 1157−1174. [doi: 10.1002/spe.4380231007]

[16] Koenig A, Stroustrup B. Exception handling for C++. In: Proc. of the USENIX. 1990. 149−176.

[17] Gosling J, Joy B, Steele G, Bracha G, Buckley A. The Java Programming Language Specification. New York: Addison-Wesley, 2015.

[18] Belyaev M, Ignatyev V. Exception analysis for errors detection in the SharpChecker static analyzer for C#. In: Proc. of the 2021 Ivannikov Ispras Open Conf. (ISPRAS). 2021. 8−16. [doi: 10.1109/ISPRAS53967.2021.00007]

[19] Fu C, Ryder BG. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In: Proc. of the 29th Int'l Conf. on Software Engineering (ICSE 2007). 2007. 230−239. [doi: 10.1109/ICSE.2007.35]

[20] Bob M. Common vulnerabilities enumeration (CVE), common weakness enumeration (CWE), and common quality enumeration (CQE): Attempting to systematically catalog the safety and security challenges for modern, networked, software-intensive systems. Association for Computing Machinery, 2019, 38(2): 9−42. [doi: 10.1145/3375408.3375410]

[21] Cui BJ. Software supply chain security faces the challenge of open source software. China Information Security, 2018, 107(11): 71−72 (in Chinese with English abstract).

[22] Xmirror. Software Supply Chain Security White Paper. 2021 (in Chinese).

[23] Chang BM. A review of exception analysis and its applications. In: Proc. of the 7th World Multi-Conf. on Systemics, Cybernetics and Informatics. 2003. 68−73.

[24] Mao CY, Lu YS. Study on the analysis and testing of exception handlings in C++ programs. Journal of Chinese Computer Systems, 2006, 27(3): 481−485 (in Chinese with English abstract).

[25] Chang BM, Choi K. A review on exception analysis. Information and Software Technology, 2016, 1−16. [doi: 10.1016/J.INFSOF.2016.05.003]

[26] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[27] Li Z, Jin XL, Zhuang CZ, Sun Z. Overview on graph based anomaly detection. Ruan Jian Xue Bao/Journal of Software, 2021, 32(1): 167−193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6100.htm [doi: 10.13328/j.cnki.jos.006100]

[28] Yi K. Compile-time detection of uncaught exceptions in standard ML programs. In: Proc. of the Int'l Static Analysis Symp. 1994. 864. [doi: 10.1007/3-540-58485-4_44]

[29] Yi K. An abstract interpretation for estimating uncaught exceptions in standard ML programs. Science of Computer Programming, 1998, 31(1): 147−173. [doi: 10.1016/S0167-6423(96)00044-5]

[30] Miné A. Relational abstract domains for the detection of floating-point run-time errors. In: Proc. of the European Symp. on Programming Languages and Systems (ESOP 2004). 2004. 2986. [doi: 10.1007/978-3-540-24725-8_2]

[31] Ramachandran J, Păsăreanu C, Wahl T. Symbolic execution for checking the accuracy of floating-point programs. ACM SIGSOFT Software Engineering, 2015, 40: 1−5. [doi: 10.1145/2693208.2693242]

[32] Yi K, Ryu S. Towards a cost-effective estimation of uncaught exceptions in SML programs. In: Proc. of the 4th Int'l Symp. on Static Analysis (SAS 1997). 1997. 98−113.

[33] Yi K, Ryu S. A cost-effective estimation of uncaught exceptions in standard ML programs. Theoretical Computer Science, 2002, 277(1−2): 185−217. [doi: 10.1016/S0304-3975(00)00317-0]

[34] Chang BM, Jo JW. Estimating exception-induced control flow for Java. In: Proc. of the Asian Symp. on Programming Languages and Systems. 2001.

[35] Chang BM, Jo JW, Yi K, Choe KM. Interprocedural exception analysis for Java. In: Proc. of the 2001 ACM Symp. on Applied Computing (SAC 2001). New York: Association for Computing Machinery, 2001. 620−625. [doi: 10.1145/372202.372786]

[36] Jo JW, Chang BM, Yi K, Choe KM. An uncaught exception analysis for Java. System Software, 2004, 72(1): 59−69. [doi: 10.1016/S0164-1212(03)00057-8]

[37] Ryu S, Yi K. Exception analysis for multithreaded Java programs. In: Proc. of the 2nd Asia-Pacific Conf. on Quality Software. 2001. 23−30. [doi: 10.5555/872019.872353]

[38] Jo JW, Chang BM. Constructing control flow graph for Java by decoupling exception flow from normal flow. In: Proc. of the Computational Science and Its Applications (ICCSA 2004). 2004. 106−113. [doi: 10.1007/978-3-540-24707-4_14]

[39] Chang BM, Jo JW, Her SH. Visualization of exception propagation for Java using static analysis. In: Proc. of the IEEE Workshop on Source Code Analysis and Manipulation. 2002. 173−182. [doi: 10.1109/SCAM.2002.1134117]

[40] Fähndrich M, Aiken A. Program analysis using mixed term and set constraints. In: Van Hentenryck P, ed. Proc. of the Static Analysis (SAS). 1997. [doi: 10.1007/BFb0032737]

[41] Chang C, Liu KS, Tan LD, Jia WC. Data flow analysis for C program based on graph model. Journal of Zhejiang University (Engineering Science), 2017, 51(5): 1007−1015, 1050 (in Chinese with English abstract).

[42] Allen FE, Cocke J. A program data flow analysis procedure. Communications of the ACM, 1976, 19(3): 137. [doi: 10.1145/360018.360025]

[43] Reps T, Sagiv M, Horwitz S. Interprocedural dataflow analysis via graph reachability. Computer Science, 2008.

[44] Mauro P, Michal Y. Dependence and data flow models. Software Testing and Analysis Process Principles and Techniques, 2007, 121−126.

[45] Jiang SJ, Xu BW, Shi L. An approach of data-flow analysis based on exception propagation analysis. Ruan Jian Xue Bao/Journal of Software, 2007, 18(1): 74−84 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/74.htm [doi: 10.1360/jos180074]

[46] Choi JD, Grove D, Hind M, Sarkar V. Efficient and precise modeling of exceptions for the analysis of Java programs. ACM SIGSOFT Software Engineering Notes, 1999, 24(5): 21−31.

[47] Chatterjee R, Ryder B. Data-flow-based Testing of Object-oriented Libraries. Technical Report, 433, Rutgers University, 2001.

[48] Robillard MP, Murphy GC. Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. on Software Engineering and Methodology, 2003, 12(2): 191−221.

[49] Fu C, Milanova A, Ryder BG, Wonnacott DG. Robustness testing of Java server applications. IEEE Trans. on Software Engineering, 2005, 31(4): 292−311.

[50] Weimer WR, Necula GC. Exceptional situations and program reliability. ACM Trans. on Programming Languages and Systems, 2008, 30(2): 1−51.

[51] Buse R, Weimer W. Automatic documentation inference for exceptions. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis (ISSTA 2008). New York: Association for Computing Machinery, 2008. 273−282.

[52] Hovemeyer D, Pugh W. Finding more null pointer bugs, but not too many. In: Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007). New York: Association for Computing Machinery, 2007. 9−14.

[53] Wu XQ, Xu ZX, Wei J. Static detection of bugs caused by incorrect exception handling in Java programs. In: Proc. of the Int'l Conf. on Quality Software. 2011. 61−66.

[54] Jiang SJ. Exception propagation analysis and its applications [Ph.D. Thesis]. Nanjing: Southeast University, 2006 (in Chinese with English abstract).

[55] González CR, Liblit B. Finding error-handling bugs in systems code using static analysis. In: Proc. of the PhD Forum of the 2011 Grace Hopper Celebration of Women in Computing. 2012.

[56] Prabhu P, Maeda N, Balakrishnan G, Ivan F, Gupta A. Interprocedural exception analysis for C++. In: Proc. of the European Conf. on Object-oriented Programming. 2011. 583−608.

[57] Lu XY, Ma RK, Wei Q. A survey of the research on symbolic execution. Industrial Information Security, 2022(1): 24−31 (in Chinese with English abstract).

[58] Wang ML, Zhang YN, Li MY, Shao S, Liu SR. Improving the accuracy of static defect analysis based on symbolic execution. Trans. of Beijing Institute of Technology, 2020, 40(4): 382−385, 395 (in Chinese with English abstract).

[59] Shao SH, Su T, Gu B, Wang Z, Yang MF. C testing framework study based on inter-procedural dynamic symbolic execution. Computer Engineering and Design, 2014, 35(8): 2746−2751 (in Chinese with English abstract).

[60] Kádár I, Hegedus P, Ferenc R. Runtime exception detection in Java programs using symbolic execution. Acta Cybernetica, 2014(21): 331−352.

[61] Tomb A, Brat G, Visser W. Variably interprocedural program analysis for runtime error detection. In: Proc. of the Int'l Symp. on Software Testing and Analysis. 2007. 97−107.

[62] Eler MM, Durelli VHS, Endo AT. Analyzing exceptions in the context of test data generation based on symbolic execution. In: Proc. of the 27th Int'l Conf. on Software Engineering and Knowledge Engineering. 2015. 346−351.

[63] Ntouskis G, Ioannidis S, Vasilakis N. Demo: Detecting third-party library problems with combined program analysis. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security. 2021.

[64] Zhou XC, Li WJ, Li SX. Researches and advances in type system. Computer Science, 2000(5): 5−13 (in Chinese with English abstract).

[65] Zhou XC, Li WJ, Li SX. A categorical model of type system $\lambda\omega\times\leqslant$. Journal of Computer Research and Development, 2002, 39(1): 68−72 (in Chinese with English abstract).

[66] Li XP, Wu NRQQG, Ma SD, Lü JH. High-order typed verifiable application system architecture modelling and its case. Ruan Jian Xue Bao/Journal of Software, 2020, 31(8): 2309−2335 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5963.htm [doi: 10.13328/j.cnki.jos.005963]

[67] Pessaux F, Leroy X. Type-based analysis of uncaught exceptions. In: Proc. of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 1999. 276−290.

[68] Allen N, Krishnan P, Scholz B. Combining type-analysis with points-to analysis for analyzing Java library source-code. In: Proc. of the 4th ACM SIGPLAN Int'l Workshop on State of the Art in Program Analysis (SOAP 2015). New York: Association for Computing Machinery, 2015. 13−18.

[69] Guzman J, Alfonso S. An extended type system for exception. In: Proc. of the ACM SIGPLAN Workshop on ML and Its Applications. 1994. 127−135.

[70] Marino D, Millstein T. A generic type-and-effect system. In: Proc. of the 2009 ACM SIGPLAN Int'l Workshop on Types in Languages Design and Implementation. Savannah, 2009.

[71] Gifford DK, Lucassen JM. Integrating functional and imperative programming. In: Proc. of the 13th ACM Symp. on Principles of Programming Languages. ACM, 1986.

[72] Lucassen JM, Gifford DK. Polymorphic effect systems. In: Proc. of the 15th Annual ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. 1988. 47−57.

[73] Leroy X, Weis P. Polymorphic type inference and assignment. In: Proc. of the 18th ACM Symp. of Principles of Programming Languages. ACM, 1991.

[74] Benton N, Buchlovsky P. Semantics of an effect analysis for exceptions. In: Proc. of the 2007 ACM SIGPLAN Int'l Workshop on Types in Languages Design and Implementation (TLDI 2007). New York: Association for Computing Machinery, 2007. 15−26.

[75] Glynn K, Stuckey PJ, Sulzmann M, Sondergaard H. Exception analysis for non-strict languages. In: Proc. of the 7th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP 2002). 2002.

[76] Koot R, Hage J. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In: Proc. of the ACM Workshop on Partial Evaluation and Program Manipulation. 2015. 127−138.

[77] Yi K, Ryu S, Pyun KH. Estimating uncaught exceptions in Standard ML programs from type-based equations. In: Proc. of the 20th Int'l Computer Software and Applications Conf. (COMPSAC 1996). 1996. 455−460.

[78] Laud P, Uustalu T, Vene V. Type systems equivalent to data-flow analyses for imperative languages. Theoretical Computer Science, 2006, 364(3): 292−310.

[79] Lebresne S. A system *F* with call-by-name exceptions. In: Proc. of the 35th Int'l Colloquium on Automata, Languages and Programming (ICALP 2008). 2008.

[80] Lebresne S. A type system for call-by-name exceptions. Logical Methods in Computer Science, 2009, 5(4): 1−25.

[81] Kechagia M, Spinellis D. Type checking for reliable APIs. In: Proc. of the 1st Int'l Workshop on API Usage and Evolution (WAPI 2017). 2017. 15−18.

[82] Kim IS, Yi K, Calcagno C. A polymorphic modal type system for Lisp-like multi-staged languages. In: Proc. of the ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 2006. 257−268.

[83] Eo H, Kim IS, Yi K. Type and effect system for multi-staged exceptions. In: Proc. of the 4th Asian Conf. on Programming Languages and Systems (APLAS 2006). 2006.

[84] Wang LS. Summary-based interprocedure analysis for C++ [MS. Thesis]. Chengdu: University of Electronic Science and Technology of China, 2017 (in Chinese with English abstract).

[85] Koshelev VK, Ignatiev VN, Borzilov AI, Belevantsev AA. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, 2017, 43: 268−276.

[86] Zhao YS. Research on symbolic analysis based static defect detection technique [Ph.D. Thesis]. Beijing: Beijing University of Posts and Telecommunications, 2012 (in Chinese with English abstract).

[87] Dong YK, Gong YZ, Jin DH. Null pointer dereference defect detected based on region-based memory mode. Acta Electronica Sinica, 2014, 42(9): 1744−1752 (in Chinese with English abstract).

[88] Tang H, Wang X, Zhang L, *et al*. Summary-based context-sensitive data-dependence analysis in presence of callbacks. ACM SIGPLAN Notices, 2015, 50(1): 83−95.

[89] Schwalm A. Exploring dynamic dispatch in rust. 2017. https://alschwalm.com/blog/static/2017/03/07/

[90] Zhou XL. A relational static semantics for type analysis in object-oriented language [MS. Thesis]. Guangzhou: Jinan University, 2020 (in Chinese with English abstract).

[91] Bolotnikov IV, Borodin AE. Interprocedural static analysis for finding bugs in go programs. Programming and Computing Software, 2021, 47(5): 344−352.

[92] Liu ZH, Zhu SF, Qin BQ, Chen H, Song LH. Automatically detecting and fixing concurrency bugs in go software systems. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021). New York: Association for Computing Machinery, 2021. 616−629.

[93] Li ZH, Wang JC, Sun MS, Lui JCS. MirChecker: Detecting bugs in rust programs via static analysis. In: Proc. of the 2021 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2021). New York: Association for Computing Machinery, 2021. 2183−2196.

[94] Qin BQ. An empirical study on the safety of real-world rust programs [Ph.D. Thesis]. Beijing: Beijing University of Posts and Telecommunications, 2021 (in Chinese with English abstract).

[95] Chatterjee K, Choudhary B, Pavlogiannis A. Optimal DYCK reachability for data-dependence and alias analysis. In: Proc. of the ACM on Programming Languages (POPL). 2018(2). 1−30.

[96] Gao YH, Yang YD, Zhang Y, Yang M. Data-flow summarization of Android library with points-to analysis. Journal of Chinese Computer Systems, 2018, 39(4): 686−693 (in Chinese with English abstract).

[97] Arteca E, Harner S, Pradel M, Tip F. Nessie: Automatically testing JavaScript APIs with asynchronous callbacks. In: Proc. of the 44th Int'l Conf. on Software Engineering (ICSE 2022). New York, 2022. 1494−1505.

[98] Liu BZ, Liu PM, Li YZ, Tsai CC, Silva DD, Huang J. When threads meet events: Efficient and precise static race detection with origins. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI 2021). 2021.

[99] Cho M, Lee SH, Hur CK, Lahav O. Modular data-race-freedom guarantees in the promising semantics. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation (PLDI 2021). 2021.

[100] Wood BP, Ceze L, Grossman D. Low-level detection of language-level data races with LARD. In: Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014). 2014.

[101] Pang JM, Zhao RC, Wang HM. The higher-order features of Haskell and their applications. Computer Science, 2005, 32(6): 167−168 (in Chinese with English abstract).

[102] Huang QY. Research on optimization of multi-UVA formation system based on continuation passing style [MS. Thesis]. Nanjing: Nanjing University of Posts and Telecommunications, 2020 (in Chinese with English abstract).

[103] Sekiguchi T, Sakamoto T, Yonezawa A. Portable implementation of continuation operators in imperative languages by exception handling. Lecture Notes in Computer Science, 2001, 217−233. [doi: 10.1007/3-540-45407-1_14]

[104] Philips L, Koster JD, Meuter WD, Roover CD. Dependence-driven delimited CPS transformation for JavaScript. In: Proc. of the 2016 ACM SIGPLAN Int'l Conf. on Generative Programming: Concepts and Experiences (GPCE). 2016.

[105] Perez DD, Le W. Predicate callback summaries. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering Companion (ICSE-C). 2017.

附中文参考文献:

[1] 国家自然科学基金委员会. 软件科学与工程学科发展战略. 北京: 中国科学院, 2021.

[3] 苏仟, 赵娆. 开源软件供应链安全风险分析与发展建议. 信息安全研究, 2022, 8(8): 831−835.

[5] 缪尚廷. 开源软件供应链安全研究. 保密科学技术, 2020(7): 37−42.

[6] 何熙巽, 张玉清, 刘奇旭. 软件供应链安全综述. 信息安全学报, 2020, 5(1): 57−73.

[7] 肖广娣, 叶润国, 焦程鹏. 我国软件供应链安全问题分析及对策研究. 信息技术与标准化, 2020(6): 49−52.

[8] 董国伟. 软件供应链安全现状分析与对策建议. 中国信息安全, 2021(10): 34−37.

[21] 崔宝江. 软件供应链安全面临软件开源化的挑战. 中国信息安全, 2018, 107(11): 71−72.

[22] 悬镜安全. 软件供应链安全白皮书(2021). 2021.

[24] 毛澄映, 卢炎生. C++程序中异常处理的分析与测试技术研究. 小型微型计算机系统, 2006, 27(3): 481−485.

[26] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80−109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[27] 李忠, 靳小龙, 庄传志, 孙智. 面向图的异常检测研究综述. 软件学报, 2021, 32(1): 167−193. http://www.jos.org.cn/1000-9825/6100.htm [doi: 10.13328/j.cnki.jos.006100]

[41] 常超, 刘克胜, 谭龙丹, 贾文超. 基于图模型的 C 程序数据流分析. 浙江大学学报(工学版), 2017, 51(5): 1007−1015, 1050.

[45] 姜淑娟, 徐宝文, 史亮. 一种基于异常传播分析的数据流分析方法. 软件学报, 2007, 18(1): 74−84. http://www.jos.org.cn/1000-9825/18/74.htm [doi: 10.1360/jos180074]

[54] 姜淑娟. 异常传播分析技术及其应用研究 [博士学位论文]. 南京: 东南大学, 2006.

[57] 芦笑瑜, 麻荣宽, 魏强. 符号执行研究综述. 工业信息安全, 2022(1): 24−31.

[58] 王眉林, 张旖旎, 李明月, 邵帅, 刘湿润. 基于符号执行提高缺陷分析的准确性研究. 北京理工大学学报, 2020, 40(4): 382−385, 395.

[59] 邵巳航, 苏亭, 顾斌, 王政, 杨孟飞. 基于过程间动态符号执行的 C 语言测试框架. 计算机工程与设计, 2014, 35(8): 2746−2751.

[64] 周晓聪, 李文军, 李师贤. 类型系统的研究与进展. 计算机科学, 2000(5): 5−13.
[65] 周晓聪, 李文军, 李师贤. 类型系统$\lambda\omega x\leqslant$的范畴论模型. 计算机研究与发展, 2002, 39(1): 68−72.
[66] 李小平, 乌尼日其其格, 马世龙, 吕江花. 高阶类型化可验证应用系统体系结构建模及案例. 软件学报, 2020, 31(8): 2309−2335. http://www.jos.org.cn/1000-9825/5963.htm [doi: 10.13328/j.cnki.jos.005963]
[84] 王留帅. 基于函数摘要的 C++过程间静态分析研究 [硕士学位论文]. 成都: 电子科技大学, 2017.
[86] 赵云山. 基于符号分析的静态缺陷检测技术研究 [博士学位论文]. 北京: 北京邮电大学, 2012.
[87] 董玉坤, 宫云战, 金大海. 基于区域内存模型的空指针引用缺陷检测. 电子学报, 2014, 42(9): 1744−1752.
[90] 卓习龙. 一种面向对象语言中类型分析的关系型静态语义方法 [硕士学位论文]. 广州: 暨南大学, 2020.
[94] 秦伯钦. 对现实 Rust 应用程序安全性的实证研究 [博士学位论文]. 北京: 北京邮电大学, 2021.
[96] 高颖慧, 杨亚东, 张源, 杨珉. 使用指向分析的安卓库函数数据流摘要方法. 小型微型计算机系统, 2018, 39(4): 686−693.
[101] 庞建民, 赵荣彩, 王怀民. Haskell 语言的高阶特性及其应用. 计算机科学, 2005, 32(6): 167−168.
[102] 黄秋月. 基于延续传递风格的多无人机编队系统优化研究 [硕士学位论文]. 南京: 南京邮电大学, 2020.

葛丽丽(1996−), 女, 硕士生, CCF 学生会员, 主要研究领域为程序分析.

薛渝川(1999−), 男, 硕士生, 主要研究领域为程序分析.

帅东昕(1997−), 女, 硕士生, 主要研究领域为程序分析.

杨嘉毅(1997−), 男, 硕士生, 主要研究领域为程序分析, 函数式编程.

谢金言(1998−), 男, 硕士生, 主要研究领域为程序分析.

密杰(1993−), 男, 硕士生, 主要研究领域为程序分析.

张迎周(1978−), 男, 博士, 教授, CCF 高级会员, 主要研究领域为软件工程, 程序分析.

卢跃(1995−), 男, 硕士生, 主要研究领域为程序分析, 机器学习.