Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2021-CJ-002

2021-CJ-002

高精度的大规模程序数据竞争检测方法

高凤娟，王豫，周金果，徐安孜，王林章，吴荣鑫，张川，苏振东

Technical Report 2021

软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(7):2039−2055 [doi: 10.13328/j.cnki.jos.006260]

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

高精度的大规模程序数据竞争检测方法*

高凤娟[1], 王 豫[1], 周金果[2], 徐安孜[1], 王林章[1], 吴荣鑫[3], 张 川[4], 苏振东[5]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

[2](蚂蚁集团,广东 深圳 518000)

[3](厦门大学 信息学院,福建 厦门 361005)

[4](Department of Computer Science and Engineering, The Hongkong University of Science and Technology, Hongkong, China)

[5](Department of Computer Science (ETH Zurich), Zürich, Switzerland)

通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

摘 要: 随着技术的不断发展,软件系统的非确定性(uncertainty)不断增强,数据竞争是并发系统这一类典型的非确定性软件系统中常见的缺陷.尽管数据竞争静态检测近年来取得了巨大进展,但其面临的重要问题仍然存在.先前的静态技术要么以分析精度为代价达到高扩展性,要么由于高精度分析而导致可扩展性问题.提出一种解决上述矛盾的分段分析方法——GUARD.它首先基于程序值流进行轻量级上下文敏感的数据访问分析,以识别出候选的数据竞争子路径而非完整的程序路径.接下来,进行可能并行执行(may-happen-in-parallel,即 MHP)分析来确定程序中的两个数据访问操作是否可能会同时执行.MHP 分析基于线程流图(TFG)将线程信息进行编码以便于高效地查询各个子路径之间的并发关系.最后,对于每条存在 MHP 数据访问的子路径,进行重量级路径敏感分析以确定数据竞争路径的可行性.针对 12 个开源项目的实验评估显示,GUARD 能够在 1 870s 内完成对 130 万行代码的工业规模项目的检测,且平均误报率为 16.0%.此外,GUARD 的分析速度更快,比现有的前沿技术平均快了 6.08 倍,并且显著降低了误报率.除此之外,GUARD 在其中还发现了 12 个数据竞争漏洞.将它们全部报告给了开发者,其中 8 个已得到了确认.

关键词: 数据竞争;MHP 分析;静态分析

中图法分类号: TP311

中文引用格式: 高凤娟,王豫,周金果,徐安孜,王林章,吴荣鑫,张川,苏振东.高精度的大规模程序数据竞争检测方法.软件学报, 2021,32(7):2039−2055. http://www.jos.org.cn/1000-9825/6260.htm

英文引用格式: Gao FJ, Wang Y, Zhou JG, Xu AZ, Wang LZ, Wu RX, Zhang C, Su ZD. High-precision data race detection method for large scale programs. Ruan Jian Xue Bao/Journal of Software, 2021,32(7):2039−2055 (in Chinese). http://www.jos.org.cn/1000-9825/6260.htm

High-precision Data Race Detection Method for Large Scale Programs

GAO Feng-Juan[1], WANG Yu[1], ZHOU Jin-Guo[2], XU An-Zi[1], WANG Lin-Zhang[1], WU Rong-Xin[3], ZHANG Charles[4], SU Zhen-Dong[5]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](ANT Group, Shenzheng 518000, China)

---

* 高凤娟和王豫为共同第一作者,作者顺序依据姓氏首字母排序.

* 基金项目: 国家自然科学基金(62032010)

Foundation item: National Natural Science Foundation of China (62032010)

本文由“面向非确定性的软件质量保障方法与技术”专题特约编辑陈俊洁副教授、汤恩义副教授、何啸副教授以及马晓星教授推荐.

收稿时间: 2020-09-05; 修改时间: 2020-10-26; 采用时间: 2020-12-14; jos 在线出版时间: 2021-01-22

[3](School of Informatics, National Demonstrative Software School (Xiamen University), Xiamen 361005, China)
[4](Department of Computer Science and Engineering, The Hongkong University of Science and Technology, Hongkong, China)
[5](Department of Computer Science (ETH Zurich), Zürich, Switzerland)

Abstract: With the development of techniques, the uncertainty in software systems is continuously increasing. Data race is a typical bug in current programs, which is a classic type of uncertainty programs. Despite significant progress in recent years, the important problem of practical static race detection remains open. Previous static techniques either suffer from a high false positive rate due to the compromise of precision, or scalability issues caused by a highly precise analysis. This paper presents GUARD, a staged approach to resolve this paradox. First, it performs a lightweight context-sensitive data access analysis, based on the value flow of a program, to identify the candidate data race subpaths instead of the whole program paths. Second, may-happen-in-parallel (MHP) analysis is employedto identify whether two data accesses in a program may execute concurrently. This stage is scalable, due to the design of the thread flow graph (TFG), which encodes thread information to query MHP relationship of the subpaths. Finally, for each subpath whose two data accesses are MHP, the heavyweight path-sensitive analysis is appliedto verify the feasibility of the data races. The evaluation demonstrates that GUARD can finish checking industrial-sized projects, up to 1.3MLoC, in 1 870s with an average false positive rate of 16.0%. Moreover, GUARD is faster than the state-of-the-art techniques with the average speedup 6.08X and significantly fewer false positives. Besides, GUARD has found 12 new race bugs in real-world programs. All of them are reportedtothe developers and 8 of them have been confirmed.

Key words: data race; MHP analysis; static analysis

1 引 言

随着互联网、物联网、云计算等新计算平台、新应用模式以及智能化等新软件模式的广泛运用,软件系统内外各种来源的非确定性(uncertainty)不断增强.如何面向非确定性,保障相关软件系统的质量,逐渐成为了国内外学术界关注的焦点.从软件系统内部的不确定性来看,并发程序是一类典型的非确定性软件程序.并发程序由于其随机性高,容易导致难以调试的并发缺陷.其中,数据竞争就是一种典型的并发缺陷.数据竞争发生在两个或以上的线程并发地访问相同的内存位置,并且其中至少一个是写访问.此外,这些线程未采取任何机制来防止访问操作同时进行[1].数据竞争带来的影响从轻微的内存数据损坏到许多并发相关的缺陷,包括原子性及顺序违反.例如,在 2018 年共公开了 52 个数据竞争相关的 CVE[2].数据竞争需要经过复杂的线程切换才能显露,因此它是软件系统中最难检测的漏洞之一.随着近年来软件并行化程度不断提高,竞争检测技术正变得前所未有地重要.尽管已经取得了巨大的研究进展[3−8],但工业级的静态数据竞争检测技术仍远远不能令人满意.

数据竞争可以通过静态和动态方式来检测.动态分析[3,9−12]能够检测出程序中实际存在的漏洞,但难以推断线程间存在的所有可能的交织情况.静态工具[4,5,8,13]通常能够达到很高的代码覆盖,但现有的前沿静态数据竞争检测技术在精度和可扩展性之间陷入了两难.一类方法[5,6,8,13−15]通过放弃上下文或路径敏感达到了很高的可扩展性,但这不可避免地带来了高误报率的问题.例如,本文对一种近期提出的路径敏感的数据竞争检测技术 LDruid[8]进行了评估,它的误报率在 70%以上.另一类方法[4,5]追求高精度,但不可避免地存在可扩展性方面的问题.如 LOCKSMITH[4]为程序生成部分约束并求解,这限制了它的可扩展性.本文的实验结果表明,它无法在 1 小时内完成对 1 万行以上规模项目的分析.

本文解决这种矛盾的思路是基于这样一种观察:在实际程序中,使用部分程序路径来识别数据竞争就已足够了.尽管这可能会引入误报(例如程序入口到该路径的约束不可达),但因为对于每个数据竞争都至少存在一条局部路径,所以不会产生漏报.相比之下,传统的竞争检测方法[4−6,8]会分析整个程序路径(即从程序入口点开始的路径)来检测竞争,而对整个程序路径进行路径敏感分析是代价高昂的.所以,本文只对以两次变量访问为起止的子路径进行分析,因为这样的子路径是重要的数据竞争候选路径.

由于子路径总体上比整个程序路径更简单、明了,实行重量级分析,例如流敏感、上下文敏感、路径敏感分析是可行的.但是这些分析技术代价很高,路径敏感分析尤为如此,因为路径条件可能会错综庞杂且计算成本高昂[16].因此,现有的静态数据竞争检测技术都不能同时实现流、上下文和路径敏感分析.为了解决重量级分析(即路径敏感分析)所带来的分析开销问题,我们设计了一种分段分析的方法.首先进行值流分析[17,18]识别出候选的数据竞争子路径,接着进行可能并行执行(may-happen-in-parallel,简称 MHP)分析筛除不可行的竞争子路径.最

后,对于可能并发的数据访问子路径,采用一种更精确且更重量级的方法求解路径的可行性,以此来实现路径敏感.

MHP 分析对于 GUARD 的总体性能是至关重要的,这需要以很高的精度识别非并发的访问对.然而,前沿的 MHP 分析技术[8,19−23]面临着精度问题.例如,大多数工作[19−23]聚焦于简单的线程模型(即缺少丰富的并行化模型),且都未考虑路径约束.为了克服这种局限性,GUARD 引入了一种称作线程流图(TFG)的线程信息表示形式,用于高效且精确地编码并发信息.TFG 中的每个节点按线程标签总结了其过程内的 MHP 信息.为了最小化计算开销,线程标签将每个线程操作(例如 *thread_create*)编码为一个三元组.根据实验结果,这一步会筛除大约 99.5% 的数据访问对.为了保证精度以及可扩展性,MHP 分析和数据访问分析都采用了一种自定义的约束求解器来筛除“简单但矛盾”的路径,以此进一步减少不会导致数据竞争的路径.最后,GUARD 对于剩余的候选项调用一个完整的 SMT 求解器(例如 Z3).基于该分段分析方法,GUARD 能够精确地为程序行为建模,同时也保证了效率.

为了对 GUARD 的有效性以及效率进行评估,我们选取 12 个现实世界中的开源项目.评估工作显示 GUARD 能够在 1 870s 内完成对 130 万行代码的工业规模项目的检测,平均误报率为 16.0%.并且,GUARD 比现有的前沿技术平均快 6.08 倍,与此同时取得了更低的误报率及更高的召回率.

本文的贡献总结如下:

- 提出了 GUARD,一种检测数据竞争的分段分析方法.GUARD 与其他前沿技术相比具有更高的扩展性及精度.
- 评估了 GUARD 在多线程测试基准以及现实世界 C/C++应用程序上的性能,结果表明,它能够在短时间内以可观的精度检测数据竞争.
- GUARD 已经发现 12 个真实程序中存在的漏洞,并且其中的 8 个已得到确认.

本文第 2 节通过一个示例程序阐述 GUARD,并介绍 GUARD 的分析基础.第 3 节详细描述 GUARD.第 4 节、第 5 节为实现和实验,评估 GUARD 的效率和有效性.第 6 节介绍相关工作.最后第 7 节给出结论.

2 GUARD 概览和分析基础

本节从一个示例程序开始,结合 GUARD 框架介绍 GUARD 各个模块的作用.之后再介绍整个分析过程基于的线程模型.最后是提出所需的基础结构,即线程流图(TFG).

2.1 启发性案例

图 1 是一个示例程序.图中函数 foo 创建了一个线程执行函数 writer(第 4 行).接着,它将变量 arg 自增(第 7 行),然后等待线程结束(第 8 行).最后,它再一次将 arg 自增.在第 14 行和第 7 行存在数据竞争,因为函数 writer 和 arg++指令能够并行执行.但是第 14 行和第 9 行不存在数据竞争,因为 writer 线程在第 8 行被执行了等待终止操作.此外,静态分析工具在报告数据竞争漏洞时也应附带路径信息作为佐证,在此例中可能发生数据竞争的路径是第 4 行~第 14 行、第 6 行~第 7 行.

```
1  void foo(bool ifInc) {
2    pthread_t tid;
3    int arg = 0;
4    thread_create(&tid, writer, &arg);
5    // Increment arg by 1 if ifInc is true.
6    if (ifInc)
7      arg++; // race instruction
8    thread_join(tid);
9    arg++;
10 }
11
12 void* writer(void *arg) {
13   // It writes non-zero value.
14   write(arg); // race instruction
15 }
```

Fig.1 A motivating example

图 1 示例程序

总的来说,检测数据竞争漏洞存在许多挑战.第 1 个挑战在于检查数据访问是否可能并行发生.第 2 个挑战在于通过识别出不可行路径来消除误报.通过分析路径约束,可以避免在分析现实世界应用程序时产生误报.与此同时,静态分析工具应该提供路径信息以指明每条数据竞争可能的执行轨迹.与此相对,若工具只提供了所有可能的执行路径,而没有考虑路径约束,则需要更多的人力来确认数据竞争警报.最后一个挑战是由复杂的线程操作带来的,这些操作为程序引入了不同的线程语义信息.在本文的示例中使用到 *thread_create* 和 *thread_join* 这两个典型的线程操作.除此之外,GUARD 还建模了 4 个线程操作:*thread_wait*、*thread_notify*、*thread_lock*、*thread_unlock*.不同的线程操作会引入不同的线程语义,这使得 MHP 分析变得更为复杂.

2.2 GUARD框架

为了更清晰地阐述 GUARD 的原理,本文将该方法分为 4 个组件.图 2 展示了 GUARD 及其 4 个组件的概况:(1) TFG 构建.构建一个线程流图(TFG),用于表示整个程序的线程信息.(2) MHP 分析.利用 TFG 综合出 MHP 信息,为进一步的 MHP 查询作准备.(3) Source-Sink 框架.利用预定义的 source-sink 模式检测漏洞,且根据值流找出从一次数据访问开始到另一次数据访问结束的子路径.(4) 数据竞争检查器.为数据竞争定义 source-sink 模式,以此检测数据访问子路径.对于每条数据访问子路径,GUARD 查询 MHP 关系以避免误报.换言之,它会过滤掉无法并发访问相同内存位置的数据访问对.最后,为了通过路径敏感达到更高精度,GUARD 调用约束求解器来验证 MHP 分析和数据竞争检测器所得到的路径的可行性.

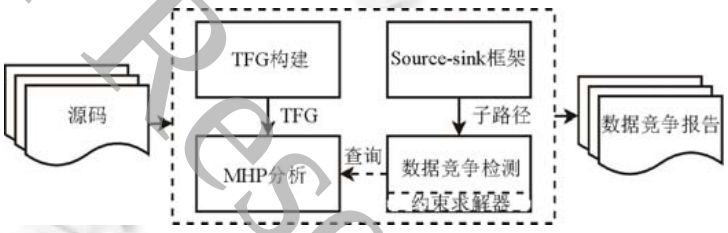

Fig.2 Overview of GUARD

图 2 GUARD 框架

TFG 构建.这一部分会构建 TFG.TFG 是一个全局的图结构,它连接了不同的函数调用、线程操作以及控制流.GUARD 直接从程序指令构建函数级的 TFG,TFG 中的节点根据函数调用及控制流来构造.接着,基于线程操作及函数调用信息连接各个函数级的 TFG.这与控制流图(CFG)的构建方式有些类似,所不同的是,GUARD 为线程操作设计了一些特殊的语义,而不是将它们当作普通的函数调用进行处理.例如,对于一个 *thread_create* 函数调用,GUARD 在 TFG 中创建一条从调用点指向被创建线程入口的边.另外,根据函数调用,GUARD 将一系列指令(CFG 中的一个块)分割为 2 个或以上的节点,这是因为,在一次函数调用前后,MHP 关系可能会有所不同.与 CFG 相比,TFG 拥有 3 个特性:CFG 中的多个基本块如果有相同的 MHP 关系就可以合并为 TFG 中的一个节点;一个块中的任意函数调用或线程操作都会将该块分为两个节点,且以函数调用或线程操作为分界;各个函数级的 TFG 通过函数调用或线程操作连接在一起.因此,TFG 关注线程相关的操作以及部分控制流,而非仅仅是控制流.上述特征使得 MHP 信息的计算更为高效.图 3 是对图 1 中案例的简化分析过程.该 TFG 包含 2 个函数级 TFG,它们的入口节点以其函数名为开始,出口节点用(0)标出.图 3(a)是示例所对应的 TFG,节点中的 $(l_i,\ldots,l_j)$ 代表示例中相应的代码行.因为 foo 中创建了一个新的执行 writer 函数的线程,所以 foo 所对应的 TFG 中的节点(2,3,4)连到 writer 对应的 TFG 入口节点.节点(6,7,8)被 writer 所对应的 TFG 的出口节点所指向,这是因为,它对线程执行了 join 操作.值得注意的是,由于第 6 行和第 7 行的分支拥有相同的 MHP 关系,TFG 将它们合并为了一个节点.除此之外,在第 4 行和第 8 行的函数调用前后 MHP 关系发生了改变,于是 GUARD 根据这两个函数调用将程序指令划分为了不同节点.其后继节点(即 node(9))包含指令 arg++.鉴于函数可能存在多条 return 指令,每个函数中的终节点将函数的所有出口点进行了合并.总体而言,构建 TFG 的目的在于实行精确的 MHP 分析并且避免冗余分析.

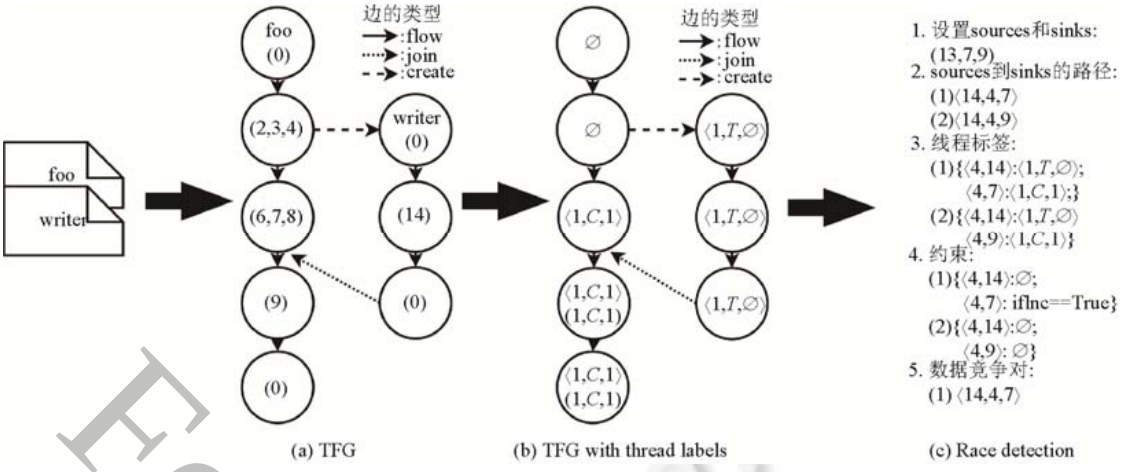

Fig.3 Simplified analysis procedures for Fig.1

图 3 图 1 示例的简化分析过程

MHP 分析.GUARD 通过为 TFG 节点构建线程标签来表示 MHP 信息.首先提取出线程相关的信息(即 *thread_create* 和 *thread_join*),并用线程标签对它们建模.然后基于线程标签分析两条路径的 MHP 关系.图 3(b)中展现了每个节点简化的线程标签.初始标签集是∅,然后 *thread_create* 函数创建了一个新的线程标签.线程标签包含两种标签,一类以⟨1,*T*,∅⟩表示,分配给刚创建的线程.它所对应的标签是⟨1,*C*,1⟩,在 *thread_create* 调用点之后被赋给节点.在 *thread_join* 的调用点处,添加一个标签(1,*C*,1),表明这个标签执行了等待线程终止这个操作.线程标签中的第 1 个元素是 *thread_create* 的标识,第 2 个元素表明了标签类型,最后一个元素用于区分不同的调用上下文.⟨⟩和()分别表示 *thread_create* 和 *thread_join* 所对应的标签集.有关线程标签的详细讨论见第 3.1.1 节.

Source-Sink 框架.该框架基于值流分析(value flow analysis),自动依据与特定缺陷对应的 source-sink 模式分析缺陷.很多软件缺陷都可以被表达为如下的形式:在任意一个程序中,一个由程序事件 *A* 生成的值 *v* 一定能够流向另一个程序事件 *B* 中的一个值[24].所以,该框架检测任何一个能够被建模成基于 source 和 sink 模式的全局值流的缺陷.并且,检测出的由 source 到 sink 的路径并不会由程序入口开始.

数据竞争检查器.如图 3(c)所示,最后一步是竞争检测,这依赖于 Source-Sink 框架以及 MHP 分析.首先,它会基于 source-sink 框架检测读-写或写-写对.source 和 sink 都被设定为对变量的读或写操作,因此第 7 行、第 9 行和第 14 行的变量 arg 同时为 sources 和 sinks.接着,GUARD 分析程序的值流,找到从 source 到 sink 的路径.在这一步中,检测到的 source-sink 路径是⟨14,4,7⟩和⟨14,4,9⟩.然后根据这两条路径的子路径(即{⟨4,7⟩,⟨4,14⟩}和{⟨4,9⟩,⟨4,14⟩})提取出线程标签以及路径约束.最后一步是根据它们所对应的线程标签和路径约束检测数据竞争.由于第 4 行创建的线程在第 8 行执行了等待结束操作,⟨14,4,9⟩并不是 MHP 的,因此对于启发性示例,GUARD 只会报告一个数据竞争对⟨14,4,7⟩.

前文展示的启发性示例描述了 GUARD 检测数据竞争的基本思路.接下来,本文将介绍有关检测实际程序数据竞争时的分析细节.

2.3 线程模型

图 1 的示例程序中只包含两种线程操作:*thread_create* 和 *thread_join*.现代编程语言提供了种类繁多的线程操作来实现不同的并行模式.本文考虑如下 6 种具有代表性的函数.

- *thread _ create*(*tid*,*routine*, *para*) 接受 3 个参数:一个线程标识号(线程 ID)、一个指向所要执行任务的指针及其所需的单个参数值.
- *thread _ join*(*tid*) 等待 *tid* 所指明的线程终止.
- *thread _ notify*(*c*) 唤醒执行了 *thread _ wait*(*c*) 的线程.

- *thread _lock*(*mutex*) 和 *thread _unlock*(*mutex*) 给 *mutex* 加锁或去锁.

2.4 线程流图

GUARD 首先生成函数级 TFG,并根据线程操作和函数调用将函数级 TFG 连接起来构建起一个全局 TFG.

TFG 定义为 $\mathcal{G}=(\mathcal{N},\mathcal{E})$, 它包含一个点集 $\mathcal{N}=\{n_1,\ldots,n_m\}$ 以及一个由有向边组成的边集 $\mathcal{E}=\{E_1,\ldots,E_K\}$, 其中,$K$ 是边的种数,E_k 是类型为 k 的边的集合.在一个函数级 TFG 中,每个节点由指令序列构成,彼此间以控制流指令或函数调用为分割.为了连接各节点,TFG 引入用于指明控制流的边$\rightarrow_{flow}$.与 CFG 类似,TFG 也有一个入口节点和一个出口节点(在 $\mathcal{N}$ 中).入口节点没有入边而出口节点没有出边.

在一个全局 TFG 中,函数调用中的调用者和被调用者由 6 种边连接,分别是:$\rightarrow_{call}$、$\rightarrow_{create}$、$\rightarrow_{join}$、$\rightarrow_{notify}$、$\rightarrow_{lock}$和$\rightarrow_{unlock}$.其中,$\rightarrow_{create}$将 *thread_create* 的调用点连接至线程的入口点;$\rightarrow_{join}$将线程的出口点连接至 *thread_join* 的调用点;*thread_notify* 的调用点通过$\rightarrow_{notify}$与相应的 *thread_wait* 的调用点连接起来;从 n_1 到 n_2 的同步边(即$\rightarrow_{notify}$)表明:一旦 *thread_wait* 在 n_2 处执行,n_2 必定会接收到来自 n_1 的消息;*thread_lock* 或 *thread_unlock* 的调用点通过$\rightarrow_{lock}$ 或$\rightarrow_{unlock}$ 连接至其后继节点.其他函数调用通过$\rightarrow_{call}$ 将它们的调用点连接至对应的被调用者的入口点.除此之外,后 5 种边需要通过线程标识符(thread ID)匹配对应的操作,例如 *thread_join* 需要匹配与其参数(即线程标识符)对应的,且由 *thread_create* 创建的线程.这一部分通过值流分析实现(见第 3.2 节).

从 TFG 中可以看出,同一节点中的所有指令拥有相同的 MHP 关系.此外,相邻节点可能也具有相同的 MHP 关系.为了提高 MHP 分析的效率,GUARD 根据算法 1 将具有相同 MHP 信息的邻接节点合并为一个.对于按拓扑序排列的每个函数级 TFG(第 1 行),首先用函数调用对 TFG 节点进行标记(第 3 行).allMinimalLoopOrBranch 抽取出循环及分支所对应的节点.如果存在嵌套的循环或分支,它只获取最深层的循环或分支.换言之,给定一个嵌套的分支或循环 $\mathcal{L}=\{l_1,\ldots,l_n\}$, 其中,$l_i$ 包含点集 $\mathcal{N}_l^i$, 分析出 l_j s.t. $\forall l_i\in\mathcal{L},\mathcal{N}_l^j\in\mathcal{N}_l^i$. 之后对于 allMinimal LoopOrBranch 抽取出的每个循环或分支,对其相应的节点进行标记(第 5 行~第 8 行).在标记节点之后,基于循环或分支对节点进行合并.该算法一直合并节点,直到没有节点可以被合并(第 10 行~第 14 行).

算法 1. Compressing Thread-flow Graph.

Input: TFG $\mathcal{G}$;

Output: Compact TFG $\mathcal{G}'$.

(1) **for** $\mathcal{G}_f'$ in reversedTopologicalOrder($\mathcal{G}$) **do**

(2) //It marks function calls, including thread operations.

(3) markNodeWithFunctionCall($\mathcal{G}_f$);

(4) $\mathcal{B}$ =allMinimalLoopOrBranch($\mathcal{G}_f$);

(5) **for** $\mathcal{L}\in\mathcal{B}$ do

(6) **if** $\exists v\in\mathcal{L}$, is $Marked(v)==$ true **then**

(7) **for** $v\in\mathcal{L}$ **do**

(8) markNode(v);

(9) $\mathcal{G}_f'=\mathcal{G}_f$;

(10) **while** $\mathcal{G}_f'$ is unchanged **do**

(11) $\mathcal{B}$ =allMinimalLoopOrBranch($\mathcal{G}_f'$);

(12) **for** $\mathcal{L}\in\mathcal{B}$ do

(13) **if** $\forall v\in\mathcal{L}$, is $Marked(v)==$ false **then**

(14) mergeNodes($\mathcal{L}$);

3 GUARD 分析方法

本节将展示 GUARD 分析一个程序时的详细步骤.首先,具体阐释 GUARD 如何通过 TFG 分析 MHP 关系.之后,基于 source-sink 检测出数据访问对,并根据 MHP 分析筛除不可行路径.最后,通过约束求解器求解路径约

束并得到最终的数据竞争警报.

3.1 MHP分析

GUARD 应能快速、准确地查询 query 关系以减少潜在的数据竞争数量.为了实现这一点,我们设计了线程流图(TFG),用于为线程操作建模.基于 TFG,将 MHP 关系编码为线程标签,并使用转换规则模拟由线程操作所引入的语义信息.最后,根据 TFG 以及转换规则总结出 MHP 关系.

3.1.1 线程标签流

在构建完 TFG 之后,GUARD 使用线程标签及预设的转换规则来建模线程操作所引入的语义信息.首先,线程标签的定义如图 4 所示.*CT* 是一个调用序列,它由一系列调用点 CS_i 拼接而成, $\mathcal{L}$ 是 *L* 的集合,其中,*L* 是由一个三元组 $\langle ID,T,CT\rangle$ 表示的线程标签.ID 指明何种线程操作,*T* 表示标签种类,可以是 *Thread*、*CS* 或 *Other*. *Thread* 意味着该标签是由 *thread _create* 创建的,并且 *CS* 是 *thread _create* 的调用点. *Other* 代表不由 *thread _create* 创建的标签.*CT* 表示由调用点标识组成的调用轨迹.一个节点的 MHP 关系由一个五元组 $\langle NodeID,\mathcal{L}_c,\mathcal{L}_j,\mathcal{L}_{wn},\mathcal{L}_l\rangle$ 表示. *NodeID* 是当前节点的标识, $\mathcal{L}$ 是线程标签集. $\mathcal{L}_c$、$\mathcal{L}_j$、$\mathcal{L}_{wn}$、$\mathcal{L}_l$ 分别代表 *thread _create* 、 *thread _ join* 、*thread_wait*、 *thread _notify* 和 *thread _lock* 所对应的线程标签集.

$$
\begin{aligned}
CT &:= \|_{i=1}^{n} CS_i \\
L \in \mathcal{L} &:= < ID, T, CT > \\
&\quad T \in \{Thread, CS, Other\} \\
NL &:= < NodeID, \mathcal{L}_c, \mathcal{L}_j, \mathcal{L}_{wn}, \mathcal{L}_l >
\end{aligned}
$$

Fig.4 Definition of thread labels

图 4 线程标签定义

接下来,对于每个函数,根据图 5 中的转换规则对线程标签进行转换. $TL(n)[t]$ 表示查找节点 *n* 对应的 $\mathcal{L}_t$, 其中, $t \in \{c, j, wn, l\}$. 如果 *t* 未给出,则通过 *TL*(*n*) 表示检索出节点 *n* 的 *NL*(图 4 中定义).*N*(*CS*)用于查找出具有调用点 *CS* 的节点.$N_i(CS)$和 $N_o(CS)$代表 *CS* 所在函数级 TFG 的入口点和出口点. $n_i \rightarrow_t n_j$ 表示 n_i 通过类型为 *t* 的边连接至 n_j.注意,n_i(或 n_j)可能为 *N*(*CS*)(即 $N(CS) \rightarrow_t n_j$),这时它表示一个具有调用点 *CS* 的节点通过类型为 *t* 的边连接至 n_j. $\dot{N}(CS)$ 和 $\dot{N}(CS)$ 分别表示节点 *N*(*CS*)之前和之后的点. $Succ(n_i)$ 和 $Pred(n_i)$ 表示 TFG 中节点 n_i 的后继和前驱. $Obj_i Oper = Obj_j$ 代表通过 *Oper* 的一次赋值,例如 $Obj_i \bigcup = Obj_j$ 表示合并这两个集合并复制给 Obj_i.

Flow 规则表示一次基本的线程标签转换.如果有 *k* 个节点通过 $\rightarrow_{flow}$ 与点 n_m 相连,它将来自前驱节点(例如分支)的 $\mathcal{L}_c$ 合并,并取 $\mathcal{L}_j$、$\mathcal{L}_{wn}$、$\mathcal{L}_l$ 各自的交集.如果一个节点只有一个前驱节点,其后继节点与前驱节点拥有相同的线程标签.

Call 规则用于在函数调用处转换线程标签.当一个线程标签集合到达一处函数调用时,调用点将其当前的标签集传递给被调用者,调用点的输出标签集则是合并被调用者出口节点的标签集.

Create 规则对应 *thread _create* 操作的转换规则.注意,线程的创建引入两个新的标签(L_{CS} 和 L_T),这两个标签被添加至 *NL* 中对应的 $\mathcal{L}_C$ 内.在被创建的线程上,该规则将标签传递给被创建线程并将它们合并.在 *thread_ create* 的调用点,只将 $\mathcal{L}_C$ 中的标签合并,而不是将 4 个集合中的标签都合并.这是线程创建和函数调用两个规则在标签传播上的核心区别.在一个函数调用中,被调用者可被视为一个内联函数,这意味着调用点之前的线程标签与被调用者出口节点之前的线程标签是相同的.类似地,调用点之后的线程标签和被调用者出口节点之后的线程标签是相同的.相比之下,线程的创建操作引入了不同的语义,因而不能被视为内联函数,所以 GUARD 为函数调用和线程创建定义了两种转换规则.

Join、*Wait* / *Notify*、*Lock*、*Unlock*、*CallingContext* 规则展现了剩余线程操作的线程标签流.*Join* 规则表示,如果一个标签 L_j 被等待终止,则会将标签添加至 *join* 集中. *Wait* / *Notify* 规则说明会将一个 ID(即这两种操作的唯一标识)加入 *thread _notify* 所有前驱节点以及 *thread _wait* 所有后继节点(包括过程间节点)的标签集中.

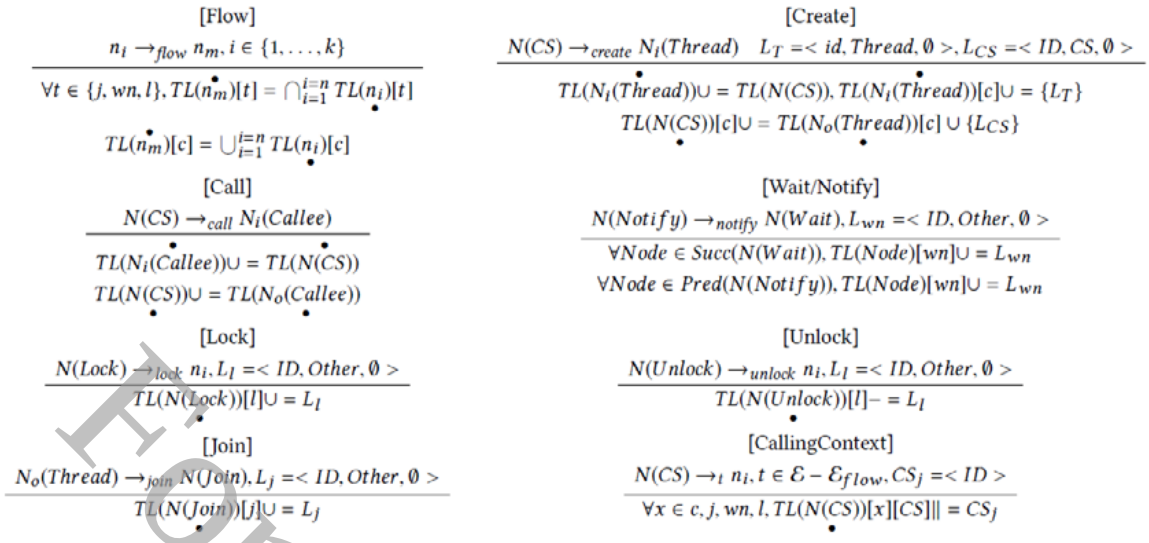

Fig.5 Transformation rules

图 5 转换规则

Lock、*Unlock* 规则展现了 GUARD 是如何分析锁的.当 *thread _lock* 函数被调用时,GUARD 将一个标签添加至锁集 L_l 中,对应地,当 *thread _unlock* 函数被调用时,它将相应锁集中的锁去除.这两条转换规则模拟了加锁和去锁的语义.

CallingContext 规则描述了 GUARD 实现上下文敏感的核心策略.对于 *NL* 中的所有标签,按调用点 ID 将每个标签中的 *CS* 进行拼接.注意,每个 *CS* 的 ID 是拼接而成的,因此 GUARD 可以从一个 *CS* 中提取出一个调用链.按照这种方式,GUARD 能够通过分析 *CS* 中的各 ID 识别出不同的调用上下文.

从图 5 可以看出,GUARD 将 *thread _ join* 或同步操作(*thread _wait* 和 *thread _notify*)分别存储在 $\mathcal{L}_j$和$\mathcal{L}_{wn}$中,而不是清除相应的线程标签.这么做的原因在于,被清除的标签可能会隐含其他函数的 MHP 关系,清除线程标签可能会带来漏报.

3.1.2 MHP 分析

在定义线程标签和转换规则后,GUARD 根据转换规则通过自底向上和自顶向下分析的方法进行 MHP 分析.自底向上分析会为每个节点计算线程标签,自顶向下分析将调用者的线程标签传递给它们的被调用者.

在自底向上分析的过程中,GUARD 以逆拓扑序分析函数.在此阶段,GUARD 根据图 5 所示的转换规则创建并更新线程标签.在此阶段后,GUARD 可以通过路径查询两条指令的 MHP 关系.但这一阶段没有路径信息 GUARD,无法高效地查询 MHP 关系,这是因为,GUARD 需要从两条指令所能到达的路径中提取出所有标签.为了不失一般性,GUARD 的目标是,无论是否有路径信息都能查询 MHP 关系,因为有些 MHP 关系的查询有路径信息而有些没有,因此 GUARD 执行第 2 阶段的自顶向下分析.

在自顶向下分析的过程中,GUARD 以函数的拓扑序分析函数标签.它将调用者的线程标签传递给被调用者,从而每个函数都能知晓其全局线程标签.通过这种方式,GUARD 总结出了全局线程标签,并且不需要路径信息就能高效地查询两条指令的 MHP 关系.

3.2 基于Source-Sink的检测框架

该框架提供了两种基础分析:一种需求驱动的、路径、上下文敏感的值流分析和一个基于 Source-Sink 的检测框架.值流分析提供了数据依赖信息,以此来识别线程操作的标识.例如,对 *thread _ join*(*tid*), *tid* 必须与被执行 *join* 操作的线程的标识相同.类似地, *thread _notify* 或 *thread _lock* 的线程标识也应该通过值流分析传递给对应的 *thread _wait* 或 *thread _unlock* .GUARD 借助别名分析的方式解决这个问题,GUARD 首先会进行完备但不

精确的指针分析,然后,给定一条特定的路径,它会收集该路径的约束条件来进行精确的指针分析.

算法 2 展示了基于 Source-Sink 的检测框架,该框架提供了对 source 和 sink 的配置.setSource、setSink 和 checkSourceSinkPair 函数是由特定的漏洞检测器定义的.setSource 定义了哪些变量是 source;setSink 定义了哪些变量为 sink;checkSourceSinkPair 则依据缺陷类型分析每条 source-sink 路径,如果该 source-sink 匹配特定的漏洞模式,GUARD 就会将其报告出来.以内存泄露检测为例,source 设为每个内存分配函数,sink 设为内存回收函数.所以每个内存分配函数(source)应该对应某条路径中的内存回收函数(sink),以此避免内存泄漏.关于数据竞争的 source 和 sink,我们在下一节给出定义.retrieveCalleeSummary 函数将被调用者的 source 和 sink 信息传递给调用者,因此 sources 和 sinks 包含了被调用者的 source 和 sink 信息.被调用者的 source 和 sink 通过参数传播(全局变量被视作参数,详见第 4.1 节),因此调用者需要能够获取被调用者已经分析的值流信息.由于 source 和 sink 可能不在同一函数中,buildPath 函数通过调用链提取出从 source 到 sink 的路径.注意,从 source 到 sink 可能不止一条路径,因此,变量 *paths* 可能会包含多条路径.最后,如果漏洞路径是可达的,那么,reportBugWithPath 就会报告该漏洞.

该框架按逆拓扑序分析程序(第 1 行),这意味着它将依据函数调用图自底向上地分析函数.在分析过程中收集 source 以及它们对应的 sink,当找到一个 source-sink 对时,该算法会检查 source 和 sink 是否都满足特定的漏洞特征模式,如果是,报告此 source-sink 对以及路径信息,否则,忽略它(第 7 行~第 12 行).可以看出,GUARD 总结 source-sink 信息(第 6 行)并检查每个 source-sink 对的可行性(第 11 行),因而该步骤是上下文敏感、路径敏感的.值得注意的是,一个 source-sink 路径 p 可以被分为两条路径 p_1 和 p_2,且 p_1 和 p_2 有相同的起点且分别在 source 和 sink 中结束,即 $p=\{p_1,p_2\}$, 其中, $p_1=(n_j,...,n_{source})$, $p_2=(n_j,...,n_{sink})$.

算法 2. Source-Sink based detection algorithm.

Input: Program;

Output: Race Bugs.

(1) **for** *fun* in reversedTopologicalOrder(Program) **do**
(2) sources=setSource(*fun*);
(3) sinks=setSink(*fun*);
(4) //The following function retrieves summarized sources
(5) //and sinks from *fun*'s callees.
(6) retrieveCalleeSummary(sources,sinks);
(7) **for** source in sources **do**
(8) **for** sink in sinks **do**
(9) //verify each sink in variable sinks.
(10) paths=buildPath(source,sink);
(11) isFeasible=checkSourceSinkPair(source,sink,paths);
(12) reportBugWithPath(source, sink, paths, isFeasible);

3.3 竞争检测

基于 MHP 分析和 source-sink 框架,GUARD 使用算法 3 通过 source-sink 模式检测数据竞争.首先,GUARD 会检查 source 和 sink 是否满足数据竞争的一个条件,即其中至少有一个写操作(第 1 行).为了避免报告读-读的数据访问对,GUARD 添加了一个访问检查(第 2 行~第 3 行).接下来,通过 isPathOrderReversed、isReachable 和 isMHP 精化竞争检测.isPathOrderReversed 检测一个 source-sink 路径(即 p)是否是逆序的,以此来消除重复的报告.GUARD 将 source 和 sink 都设为读或写,因而一个 source 可能也是一个 sink,反之亦然,这将导致,如果忽略顺序,一个漏洞可能会被报告两次.在检测完路径顺序之后,GUARD 会验证 source-sink 路径是否可达.isRechable 提取出从 source 到 sink 的路径约束并调用自定义的约束求解器筛除"简单但矛盾"的路径,之后,通过如下规则确认 source-sink 路径是否是 MHP 的.

isMHP.令 p_1 和 p_2 是来自一个给定 source-sink 路径 p 的两条路径,令 NL_1 和 NL_2 为从 p_1 和 p_2 中提取出来的节点标签.路径 p_1 和 p_2 是 MHP 的,当:

(1) $NL_1[l] \cap NL_2[l] = \varnothing$;

(2) $NL_1[wn] \cap NL_2[wn] = \varnothing$;

(3) $\forall L_{1c} \in NL_1[c], \mathcal{L}_2 = \{L_{2c} : L_{2c} \in NL_2[c]$ where $L_{2c}[ID] = L_{1c}[ID]$ and $L_{2c}[T] \neq L_{1c}[T]\}, \exists L_{2c} \in \mathcal{L}_2, L_{2c} \notin NL_1[j] \cup NL_2[j]$.

在本文所定义的线程模型下,这 3 个条件是 p_1 和 p_2 为 MHP 的充分条件,因为:(1) 它们没有被同一个锁所保护;(2) 它们没有通过 *wait* 或 *notify* 进行同步;(3) 根据它们的调用轨迹存在一个没有被 *join* 的线程.最后,通过isFeasible检测路径是否可行,它会提取出路径约束并调用一个常规的约束求解器来验证路径的可行性(第10行).对于一个数据竞争对,GUARD 验证两个路径约束.第 1 个约束来自被识别为 MHP 的线程标签,第 2 个是相应 source 和 sink 的路径.如果它们的两种路径约束都是可满足的,则 GUARD 报告这一漏洞(第 11 行).

算法 3. CheckSourceSinkPair.

```
Input: source,sink,paths;
Output: isFeasible.
(1) if source∈{write,read} and sink∈{write,read} then
(2)   if source∈{read} and sink∈{read} then
(3)     continue;
(4)   //Analyze every path in paths
(5)   for p in paths do
(6)     if isPathOrderReversed(p) or not isReachable(p) then
(7)       continue;
(8)     if not isMHP(source,sink,p) then
(9)       continue;
(10)    if not isFeasible(p) then
(11)      return true;
(12) return false;
```

4 工具实现

GUARD 是基于 LLVM3.6.2[25]使用 Z3[26]作为 SMT 求解器来实现的.GUARD 中所有的模块都由 LLVM pass 实现,LLVM pass 用于在编译器中执行程序分析、转换和优化[27].目前,GUARD 支持 C 语言中的 POSIX thread[28]以及 C++中的 std::thread[29].

4.1 方法优化

为了提高效率,GUARD 采取了多种优化方式.第 1 种是 Thread escape 分析,它排除了那些不会与其他变量同时被访问的变量.第 2 种优化是将全局变量内联为函数参数来简化分析过程.第 3 种优化是为数据竞争检测优化 source-sink 的模式.

线程逃逸(thread escape)分析.众所周知,一个程序可能包含数目繁多的变量,更不用说与之相应的变量访问.尽管 GUARD 只会分析过程间的变量(即实际参数和全局变量),但是变量的总数仍十分庞大.线程逃逸分析能够避免分析那些无法与其他变量同时被访问的变量,即 GUARD 不会分析不带线程标签的变量.因为这样的变量不可能与其他变量一起被并行访问.这个过程是保守且高效的,它能够尽快给出线程逃逸信息,同时也不会产生漏报.例如,在第 1 次线程调用前使用的变量不会与任何线程中的变量并发执行.

内联全局变量.过程内的变量包含两种类型:函数实参以及全局变量.据我们所知,这两种变量是过程内分析的两大挑战.对于能够获取函数体的所有函数,GUARD 将全局变量当作函数实参.换言之,GUARD 通过将全

局变量以函数参数在函数调用之间传播的方式,使得被调用者的分析结果能够包括全局变量.以这种方式,通过分析函数参数来分析全局变量,统一并简化了分析过程.

简化的 source-sink 模式.为数据竞争检测所设计的 source 和 sink 是对变量的读写访问操作.但是,由于读-读访问并不会导致数据竞争,所以这样的设计模式会导致冗余分析.为了避免多余的分析,GUARD 实际上将 source 设为写操作而将 sink 设为读或写操作.这样,如果一个 source 是写而 sink 是读,就不必检查 source 和 sink 的顺序.朴素的 source-sink 模式会检查 source 和 sink 的顺序,否则,当一个访问是写而另一个访问是读时就会报告两遍警告.所以,改进后的 source-sink 模式能够减少分析变量的数量以及检查 source 和 sink 顺序的时间.

4.2 程序中的循环

由于本文方法需要为两个分析步骤收集路径约束(即 MHP 查询和 source-sink 检测),必须使得两个分析步骤中的约束条件保持一致.然而,当分析一个循环时,线程标签流难以确定循环执行的次数.尽管通过对 TFG 应用转换规则能够使得 MHP 分析到达不动点,但是,由于整体分析需要结合 MHP 分析和基于 source-sink 的值流分析,计算 MHP 的定点会存在两部分路径不一致的问题.例如,有可能出现 MHP 分析中经过 2 次某个循环但是 source-sink 则是 3 次该循环.因此,尽管 GUARD 的线程流分析和检测框架能够将各个循环分开分析,我们还是采取了一个折中方案,即将一个循环展开 2 次,以此达到两个目的:(1) 避免计算定点;(2) 避免不一致的路径约束.此外,根据 Saturn[30]和 Pinpoint[17]等工具,在基于约束条件的分析过程中,展开循环是可行的,并且将循环展开 2 次是权衡精度和效率的常用举措.

5 实验评估

为了对 GUARD 进行评估,我们考虑如下研究问题.

RQ1:GUARD 检测数据竞争的效率如何?

RQ2:GUARD 检测数据竞争的有效性如何?

RQ1 评估 GUARD 方法的性能表现,RQ2 就检测到的数据竞争数量、其减少误报率的能力对 GUARD 方法的有效性进行评估.RQ1 和 RQ2 也能评估 GUARD 和其他工具相比总体性能的差异.本文实验选取 4 种可以公开获得的前沿的静态数据竞争检测工具:Relay[6]、LOCKSMITH[4]、Goblint[5]和 LDruid[8].其中,Relay 和 LOCKSMITH 是经典数据竞争检测工具,Goblint 和 LDruid 是近期提出的技术,其中,LDruid 同样采用了 MHP 分析.所有这些工具都是上下文敏感和流敏感的,且 Goblint 额外支持路径敏感的分析,这是为了处理在分支中的锁操作或可能失败的加解锁操作.相比之下,GUARD 是上下文、流、路径敏感的.

所有实验都在 Ubuntu 16.04、Intel Core i7-8700k 处理器、32GB 物理内存条件下完成.该实验将每个程序的检测时限设为 1 小时.此外,无法在限定时间内求解的约束条件会被视为可满足的.这是为了避免漏报,因为 GUARD 会丢弃不可行路径,如果将超时约束视为不可满足会导致相应的路径被丢弃,从而可能引入漏报.

5.1 评估对象

由于 GUARD 需要待测程序的源代码,因此随机选取 Github 中比较流行的开源项目作为评估对象.这些项目涵盖,如下载工具、文件系统、数据库等不同的领域.具体信息见表 1.在选定评估对象后,使用 GUARD 对它们进行分析.最后通过人工查验报告统计其效率和有效性.表 1 列出了被测程序的相关信息,第 1 列和第 2 列分别代表它们的名称和规模,最后两列则是被测程序的描述和源码地址.

为了保证公平性,我们不把通过 GUARD 检测出来的数据竞争的项目(即表 1)作为基准去评估其他工具的效果.所以,用于比较的基准程序是基于 LDruid 和 LOCKSMITH 所提供的 7 个程序.如果分析过程在 1 小时内未完成,则被认为超时.如果警报总数小于 100,我们就会手动确认数据竞争警报.否则,随机从中选取 100 个进行确认.实验结果已经公开在 figshare 中(https://figshare.com/articles/GURAR-Experiment/9724055).

Table 1 Information of subjects

表 1 对象信息

Programs	Size (KLOC)	Description	URL
Transmission	81.835	BitTorrent 客户端	https://github.com/transmission/transmission
LunchBox	20.826	C++多线程库	https://github.com/Eyescale/Lunchbox
FineDB	15.310	Nosql 数据库	https://github.com/Amaury/FineDB
sofa-pbrpc	320.743	轻量级 RPC	https://github.com/baidu/sofa-pbrpc
s3fs-fuse	16.234	基于 FUSE 的文件系统	https://github.com/s3fs-fuse/s3fs-fuse
axel	4.608	轻量级命令行下载器	https://github.com/axel-download-accelerator/axel
leveldb	25.438	一个 key-value 数据库	https://github.com/google/leveldb
zfs	491.393	一个文件系统	https://github.com/openzfs/zfs
lwan	17.145	一个网页浏览器	https://github.com/lpereira/lwan
zfs-fuse	213.402	Fuse 下的 ZFS 文件系统	https://github.com/gordan-bobic/zfs-fuse
rsyslog	124.333	一个日志处理系统	https://github.com/rsyslog/rsyslog
RedAlert	139.407	监控服务	https://github.com/alibaba/RedAlert
zcash	1 350.028	“Zerocash”协议的实现	https://github.com/zcash/zcash

5.2 GUARD的效率

表 2 中第 2 列和第 3 列展示了构建 TFG 及分析线程标签的性能信息.第 4 列和第 5 列展现了查询 MHP 的性能表现.第 4 列是平均查找时间,第 5 列是查询次数.从表 2 中可以看出一半以上的程序都能在 1s 内完成分析,所以 GUARD 是足够高效的.大多数测试程序所消耗的内存也很低.可以看到,所有程序都能在 3.6GB 的内存消耗内完成 MHP 分析.在识别 MHP 关系方面,MHP 查询所消耗的总时长的几何平均为 0.331s.其中只有 5 个程序查询 MHP 关系的时间超过 1s.通过表 2 可以得出结论:GUARD 是十分高效的.

Table 2 Analysis results of GUARD

表 2 GUARD 的分析结果

Programs	MHP build Info.		MHP query Info.			Race analysis Info.		Detection Info.	
	Time (s)	Memory (MB)	Ave. query time (μs)	#query	Time (s)	Time (s)	Memory (MB)	Static analysis	Manual checking
Transmission	<1	40	15.6	5 371	0.084	201	3 018	2	2
LunchBox	<1	60	22.0	1 003	0.022	15	1 614	2	2
FineDB	<1	5	33.2	600	0.020	14	873	1	1
sofa-pbrpc	<1	107	11.5	19 211	0.221	536	11 408	2	2
s3fs-fuse	<1	33	13.0	6 592	0.086	42	2 261	2	1
axel	<1	3	13.3	901	0.012	192	1 002	6	6
leveldb	<1	34	23.1	173	0.004	31	2 255	10	10
zfs	80	147	588.2	36 864	21.685	1 077	10 033	8	3
lwan	1	146	17.8	1 745	0.031	117	1 246	5	5
zfs-fuse	123	3 532	13 745.9	3 880	53.348	3 247	11 341	29	19
rsyslog	45	2 008	12.2	14 861	0.181	1	863	6	6
RedAlert	<1	109	4.5	221	0.001	51	5 751	1	1
zcash	<1	61 948	N/A	0	0	1 870	36 799	0	0

表 2 中第 7 行和第 8 行是程序的数据竞争检测结果.我们分析了每个程序的执行时间和内存需求.关于前 12 个程序,它们的大小从 4.6KLOC 到 491.4KLOC 不等,都在 1 小时内、11GB 内存条件下完成了分析.数据竞争检测时长的几何平均为 83.7s,内存消耗的几何平均为 2 734.0MB.最后一个程序需要相对更多的资源进行分析.可以看到,GUARD 能够在 1 870s 内消耗 36GB 内存完成对 1.3MLoC 规模程序的分析.

表 3 是不同工具所消耗的时间和内存的比较.*T* 代表总执行时间,*Mem* 是最大消耗内存.TO 意为超时(TimeOut),Crash 代表出现了段错误,OOM 表示内存不足.在计算效率提升时,我们忽略所有失败的分析结果(即 TO、Crash 和 OOM),因为它们的分析时间是未知的.从表中可以看出,在分析小规模程序时,GUARD 并不是最高效的.但是,GUARD 在所有测试程序上平均加速为 6.08 倍.代码规模在 10KLOC 以上的程序为平均分析速度的提升提供了主要的贡献.具体而言,LOCKSMITH、LDruid、Relay、Goblint 和 GUARD 分析时间的算数平均值为 55.50s、80s、3.67s、79.83s 和 25.00s,消耗内存的算数平均值为 767MB、322.8MB、49.7MB、194.3MB 和 563.8MB.由于只有 GUARD 能够分析所有程序,所以针对每个工具的平均分析提升只考虑 GUARD 和它都

能分析的程序.例如,LOCKSMITH 只能分析前 4 个程序,GUARD 分析它们的平均时间为 9.25s,平均内存消耗是 250.25MB,所以 GUARD 针对 LOCKSMITH 的分析速度提升了 5 倍,降低了 2.1 倍的内存开销.同理,GUARD 针对 LDruid、Relay、Goblint 的分析速度分别提升了 43 倍、–0.72 倍和 1.8 倍;在内存开销上,GUARD 针对 LDruid、Relay、Goblint 的内存开销增加了 0.09 倍、6.7 倍和 1.92 倍.由于 GUARD 同时引入了上下文、流、路径敏感的分析方法,所以 GUARD 的分析开销会高于较低精度的分析方法.虽然 GUARD 的内存开销只比 LOCKSMITH 低,但是,由于其分析速度超过了 3 个工具,所以 GUARD 的整体分析效率依旧是比较高的.

Table 3 Tool comparison: Efficiency

表 3 工具比较:效率

Programs	Size (KLOC)	LOCKSMITH		LDruid		Relay		Goblint		GUARD	
		T (s)	*Mem* (MB)	*T* (s)	*Mem* (MB)	*T* (s)	*Mem* (MB)	*T* (s)	*Mem* (MB)	*T* (s)	*Mem* (MB)
aget	0.83	1	325	1	29	3	17	1	27	1	42
ctrace	1.2	1	214	1	37	3	16	1	17	1	58
smtprc	3.2	60	1 081	53	151	3	26	4	53	4	268
bzip2smp	4.2	160	1 448	1 464	804	4	41	1	38	31	633
memcached	10.9	TO	NA	1 740	593	1	34	15	117	37	769
cherokee	64.5	Crash	NA	TO	NA	TO	NA	457	914	97	1 641
icecast	20.9	NA	OOM	TO	NA	8	164	Crash	NA	4	536

GUARD 在保证分析精度的前提下保持高分析效率的原因为:首先,它使用 TFG 并合并了具有相同 MHP 关系的节点,从而缩短了 MHP 分析和查询的时间;其次,为数据竞争所使用的 source-sink 模式避免了从程序入口开始分析数据竞争;最后,GUARD 只在最后一步进行了重量级路径敏感分析以验证数据竞争的可能性.但是,由于预处理部分(即构建 TFG、收集路径约束等)需要时间,GUARD 在检测小规模的项目时效率会有所降低.

总之,从实验结果可以看出 GUARD 是高效的,大部分程序都能在可接受的时间内完成分析.表 3 中对不同工具的比较表明,GUARD 在处理大规模程序时总体上比其他工具更快.

5.3 GUARD的有效性

表 2 的最后两列列出了竞争检测结果.第 9 列是静态分析结果,它展示了警报的数量.最后一列是人工核验的结果.在本次实验中,误报率的几何平均为 16.0%.此外,我们向开发者报告了 12 个不同的数据竞争漏洞,并且其中 8 个已经得到了确认,其余正在等待确认.确认的漏洞中有 7 个已被开发者修复.

表 4 列出了各种工具检测准确性的比较情况.#Warning 是竞争检测器报告的警报数量,#Bug 代表人工确认的漏洞数量.从表 4 中可以看出,GUARD 尽管不是最高效的检测工具,但其检测数据竞争的精度是最高的.在大多数情况下,5 种工具都能检测出实际存在的数据竞争,但是另外 4 种工具会产生更多的误报.GUARD 的误报率相较而言是最低的.具体而言,LOCKSMITH、LDruid、Relay、Goblint 和 GUARD 误报率的几何平均值(由于有 0,故每个值加 1,将求得几何平均值减 1 以得到最终的平均值)为 84%、80%、93%、97%和 37%.由此可见,GUARD 的误报率是最低的,而且报告的数目相对较少,确认缺陷的人工开销也相对较少.值得一提的是,除了 GUARD 和 Relay 以外,其他工具会报告包含对一个共享变量所有可能的数据访问,对这些数据方法的组合进行确认,比只确认一条包含两个数据访问的路径需要花费更多的精力.对于所有人工确认的报告,没有发现 GUARD 漏报.

Table 4 Tool comparison: Effectiveness

表 4 工具比较:有效性

Programs	LOCKSMITH		LDruid		Relay		Goblint		GUARD	
	#Warning	#Bug	#Warning	#Bug	#Warning	#Bug	#Warning	#Bug	#Warning	#Bug
aget	62	3	41	3	24	4	51	3	27	12
ctrace	3	1	2	2	40	5	16	1	6	6
smtprc	9	1	7	1	130	1	29	1	2	2
bzip2smp	60	0	149	0	154	0	41	0	3	0
memcached	NA	NA	68	1	9	1	468	1	1	1
cherokee	NA	NA	NA	NA	NA	NA	1 127	0	0	0
icecast	NA	NA	NA	NA	2 197	0	NA	NA	0	0

从结果中也可以看出,GUARD 仍然会产生误报.通过进一步检查测试用例程序,我们发现了如下导致误报

的原因.第 1 个原因是外部函数中存在位置的访问类型(读和写).第 2 个原因在于复杂的线程创建和等待终止操作.例如,我们发现一个程序在循环过程中创建一个线程列表,并通过按位和移位运算符连接这些线程.第 3 个原因是我们将求解超时的约束当作是可满足的.

因此,对真实世界开源程序的评估肯定地回答了 RQ2 的问题,即 GUARD 在检测漏洞方面是有效的.此外,对不同竞争检测工具的对比评估也回答了 RQ2——GUARD 能够以更低的误报率检测出其他工具报告的数据竞争.

5.4 讨论以及未来工作

有效性威胁.对内在有效性的主要威胁与 GUARD 实现的正确性有关.为了减轻这种威胁,我们在工具构建期间持续地对其进行测试.测试集包含 38 个回归测试用例以及 28 个实际应用程序.其中,38 个测试用例同时也用于检验其可靠性.外部有效性威胁涉及到 GUARD 对其他编程语言的通用性以及缺陷数据集的质量.由于所定义的线程模型是一个相对通用的模型,所以 GUARD 也能够基于当前线程模型扩展到其他编程语言,例如 Java.关于缺陷数据集的质量,由于目前没有找到满足本文实验的、权威的数据竞争缺陷数据集,所以我们使用了 LDruid 和 LOCKSMITH 都使用的程序作为基准程序.基准程序中的数目、缺陷特征以及已知缺陷信息都会影响缺陷检测工具的评估结果.所以,为了进一步保障评估的准确性,收集数据竞争缺陷也是未来工作之一.

相关警告.在实验中我们发现,尽管 GUARD 对于一些程序报告了很多警报,其中的一部分涉及的是相同的变量.这是由于,可能存在两条以上的路径到达同一个数据访问对,在这种情况下,GUARD 根据这个数据访问对会报告来自不同路径的警报.此外,另一种类型的相关警报是由访问相关联位置的一个变量引起的.例如,若一个基本块中存在对一个变量的 2 次写访问,那么这个变量所对应的数据竞争会被重复识别 2 次.即使这些警报的特征是相似的,开发者也必须手动检查这些相似警报.在理想状况下,开发者只需确认其中的一个,而不必每个都进行检查.一种避免产生这两种警报的有效方式是使用警报集群(warning cluster).如果具有相似模式特征的警报能被聚集起来,那么开发者只需检查其中一个典型的警报即可,从而可以避免在相似的警报上花费精力.

6 相关工作

本节调研 MHP 分析和数据竞争检测相关的方法.

MHP 分析.先前已有许多工作分析了 MHP 关系,特别是对于具有高级并发结构的语言,如 Java[21,22]和 X10[19,20,23].此外,也有工作聚焦于其他语言[8,14,31,32].X10 拥有内置的高级并发结构,如 async、atomic 和 finish,以此来简化并行程序的分析和优化.基于这种语言,Agarwal 等人[19]提出的一种 MHP 分析算法仅遍历 X10 程序的程序结构树.Sanker 等人[23]对这种算法进行了扩展,通过一种对 MHP 条件的更高效的表示形式改善了计算复杂度.Joshi 等人[31]提高了 MHP 分析的精度,通过对动态屏障间距的推理推断出语句间的顺序.Naumovich 等人[22]提出了一种面向 Java 的使用数据流分析框架的 MHP 分析技术.该技术通过构建并行执行图(PEG)来表示 Java 并发程序,并使用 PEG 推测不同指令间的 MHP 关系.Barik[21]通过牺牲精度的方式换取效率,其采用的算法基于线程创建树,从线程层面计算 MHP 关系.高层次并发结构的 MHP 分析简化了复杂性,最近的工作也分析了低层语言中的 MHP 关系.Di 等人[14]通过分析局部关系图(RRG)计算 Pthreads 程序的 MHP 关系.RRG 是基于线程敏感的 CFG 而构建的,具有相同 happens-before 性质的语句会被分到同一区域中.该工作只建模了 *thread_create* 和 *thread_join*.而 LDruid[8]方法通过静态地计算时钟向量来建立 happens-before 关系,并建模了 4 种线程操作(*thread_create*、*thread_join*、*thread_wait* 和 *thread_notify*).与之相比,GUARD 建模了 6 种常见的线程操作.尽管 LDruid 声称使用了分离的锁集分析用于竞争检测,但它也引入了锁集分析所需的额外计算开销.上述方法背后采用的主要思路与 GUARD 是相似的,即将 MHP 关系编码为专有结构体从而通过分析该结构体识别出两个语句的 MHP 关系.但是,如果没有调用上下文信息和路径敏感分析,这些方法会产生更高的误报率.据我们所知,GUARD 是第 1 个将上下文敏感和路径敏感应用于 MHP 分析和数据竞争检测的工具.

静态方法.静态方法聚焦于在不执行程序的情况下检测数据竞争,这样可以消除漏报[4−6,13,15,33−36].但是,由于缺少运行时信息,误报很难避免.Polyvios 等人[4]开发出的 LOCKSMITH 是一个针对数据竞争的上下文敏感的

相关性分析工具.它使用基于约束的技术来计算描述保护左值的锁的关联.LOCKSMITH 仅仅被应用于检测100K 行代码的程序[6].Goblint[5]将指针和值分析相结合,从而使其能够处理复杂的加锁情况.RELAY[35]是一种使用自底向上分析方法的静态可扩展算法.它采用符号执行、指针分析、锁集分析、受保护访问分析的方法产生竞争警报.IteRace[36]是一种针对 Java 并行循环的静态数据竞争检测工具,它通过专门处理 lambda 式并行循环、跟踪、汇总,实现了较低的误报率.但其只能分析并行循环.ECHO[15]能够在编写代码期间在 IDE 中实时检测数据竞争.为了能够在 IDE 中得以使用,它采用的指针分析在程序有修改时会增量分析.然而,由于对分析速度存在一定的要求,它不是上下文敏感和路径敏感的.RacerX[33]使用流敏感的过程间分析检测数据竞争以及死锁.然而,为了实现可扩展,它丢弃了一些程序信息,例如使用类型表示所有左值,这就导致了它是不完善的.据我们所知,这些静态分析技术都没有同时实现上下文、流、路径敏感,大部分技术仅实现了其中的一部分.例如,Goblint[5]在分析锁集时只实现了路径敏感分析.与先前的工作相比,GUARD 编码 MHP 关系、上下文信息、路径信息,从而构建起了一个完备、有效的数据竞争检测器.

7 结 论

本文提出了一种阶段化的数据竞争检测技术 GUARD.它首先基于值流分析检测出数据可能的数据竞争候选,之后采用 MHP 分析,基于新的程序表现形式(即线程流图)高效地筛除不可行的数据竞争路径.最后,GUARD 使用路径敏感分析保护数据竞争检测过程,进一步降低了误报率.我们通过多个程序评估了 GUARD,结果表明,它能够精确、高效地检测已知和未知的数据竞争.未来,我们计划进一步提高分析精度,同时也扩展 GUARD,使其能够处理其他种类的并发错误.

References:

[1] Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems (TOCS), 1997,15(4):391–411.

[2] CVE security vulnerabilities related to CWE (Common Weakness Enumeration) 362. 2020. https://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html

[3] Chen D, Jiang Y, Xu C, Ma X, Lu J. Testing multithreaded programs via thread speed control. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2018. 15–25.

[4] Pratikakis P, Foster JS, Hicks M. LOCKSMITH: Context-sensitive correlation analysis for race detection. ACM SIGPLAN Notices, 2006,41(6):320–331.

[5] Vojdani V, Apinis K, Rõtov V, Seidl H, Vene V, Vogler R. Static race detection for device drivers: The Goblint approach. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2016. 391–402.

[6] Voung JW, Jhala R, Lerner S. RELAY: Static race detection on millions of lines of code. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE). ACM, 2007. 205–214.

[7] Wang Y, Wang L, Yu T, Zhao J, Li X. Automatic detection and validation of race conditions in interrupt-driven embedded software. In: Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2017. 113–124.

[8] Zhou Q, Li L, Wang L, Xue J, Feng X. May-happen-in-parallel analysis with static vector clocks. In: Proc. of the Int'l Symp. on Code Generation and Optimization. ACM, 2018. 228–240.

[9] Huang J, Rajagopalan AK. Precise and maximal race detection from incomplete traces. ACM SIGPLAN Notices, 2016,51(10): 462–476.

[10] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures. ACM SIGPLAN Notices, 2013, 48(6):141–152.

[11] Machado N, Lucia B, Rodrigues L. Concurrency debugging with differential schedule projections. ACM SIGPLAN Notices, 2015,50(6):586–595.

[12] Zhang T, Jung C, Lee D. ProRace: Practical data race detection for production use. ACM SIGPLAN Notices, 2017,52(4):149–162.

[13] Blackshear S, Gorogiannis N, O'Hearn PW, Sergey I. RacerD: Compositional static race detection. In: Proc. of the ACM on Programming Languages (OOPSLA). 2018. 1–28.

[14] Di P, Sui Y, Ye D, Xue J. Region-based may-happen-inparallel analysis for C programs. In: Proc. of the 44th Int'l Conf. on Parallel Processing (ICPP). IEEE, 2015. 889–898.

[15] Zhan S, Huang J. ECHO: Instantaneous in situ race detection in the IDE. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE). ACM, 2016. 775–786.

[16] Dillig I, Dillig T, Aiken A. Sound, complete and scalable path-sensitive analysis. ACM SIGPLAN Notices, 2008,43(6):270–280.

[17] Shi Q, Xiao X, Wu R, Zhou J, Fan G, Zhang C. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM, 2018. 693–706.

[18] Sui Y, Xue J. SVF: Interprocedural static value-flow analysis in LLVM. In: Proc. of the 25th Int'l Conf. on Compiler Construction. ACM, 2016. 265–266.

[19] Agarwal S, Barik R, Sarkar V, Shyamasundar RK. May-happen-in-parallel analysis of X10 programs. In: Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. ACM, 2007. 183–193.

[20] Albert E, Genaim S, Gordillo P. May-happen-in-parallel analysis for asynchronous programs with inter-procedural synchronization. In: Proc. of the Int'l Static Analysis Symp. Springer-Verlag, 2015. 72–89.

[21] Barik R. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In: Proc. of the Int'l Workshop on Languages and Compilers for Parallel Computing. Springer-Verlag, 2005. 152–169.

[22] Naumovich G, Avrunin GS, Clarke LA. An efficient algorithm for computing MHP information for concurrent Java programs. In: Proc. of the Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE). Springer-Verlag, 1999. 338–354.

[23] Sankar A, Chakraborty S, Nandivada VK. Improved MHP analysis. In: Proc. of the 25th Int'l Conf. on Compiler Construction. ACM, 2016. 207–217.

[24] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. ACM SIGPLAN Notices, 2007,42(6):480–491.

[25] The LLVM Compiler Infrastructure. 2020. http://llvm.org

[26] Moura LD, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, 2008. 337–340.

[27] Writing an LLVM Pass. 2020. http://llvm.org/docs/WritingAnLLVMPass.html

[28] Barney B. POSIX threads programming. 2017. https://computing.llnl.gov/tutorials/pthreads/

[29] Thread—C++ Reference. 2020. http://www.cplusplus.com/reference/thread/thread/

[30] Xie Y, Aiken A. Saturn: A SAT-based tool for bug detection. In: Proc. of the Int'l Conf. on Computer Aided Verification (CAV). Springer-Verlag, 2005. 139–143.

[31] Joshi S, Shyamasundar RK, Aggarwal SK. A new method of MHP analysis for languages with dynamic barriers. In: Proc. of the 26th IEEE Int'l Parallel and Distributed Processing Symp. Workshops & PhD Forum (IPDPSW). IEEE, 2012. 519–528.

[32] Sui Y, Di P, Xue J. Sparse flow-sensitive pointer analysis for multithreaded programs. In: Proc. of the Int'l Symp. on Code Generation and Optimization. ACM, 2016. 160–170.

[33] Engler D, Ashcraft K. RacerX: Effective, static detection of race conditions and deadlocks. ACM SIGOPS Operating Systems Review, 2003,37(5):237–252.

[34] Li Y, Liu B, Huang J. SWORD: A scalable whole program race detector for Java. In: Proc. of the 41st Int'l Conf. on Software Engineering: Companion Proc. IEEE, 2019. 75–78.

[35] Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proc. of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). ACM, 2006. 308–319.

[36] Radoi C, Dig D. Practical static race detection for Java parallel loops. In: Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA). ACM, 2013. 178–190.

高凤娟(1991－),女,学士,主要研究领域为软件工程,程序语言,软件安全.

王豫(1991－),男,学士,主要研究领域为软件工程,程序语言,软件安全.

周金果(1988－),男,博士,主要研究领域为软件工程,静态分析.

徐安孜(1997－),男,学士,CCF 学生会员,主要研究领域为软件工程,软件安全.

王林章(1973－),男,博士,教授,博士生导师,CCF 杰出会员,主要研究领域为软件工程,软件安全.

吴荣鑫(1987－),男,博士,CCF 专业会员,主要研究领域为软件工程.

张川(1974－),男,博士,主要研究领域为程序分析,质量保障.

苏振东(1970－),男,博士,博士生导师,主要研究领域为程序语言,程序分析.