软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

Go 语言程序的内存性能与安全问题实证研究*

李清伟[1], 丁伯尧[1], 张 昱[1], 陈金宝[1]

1(中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230026)
通讯作者: 张昱, E-mail: yuzhang@ustc.edu.cn

摘 要: Go 语言（Golang）作为一门新兴编程语言，利用编译时的逃逸分析与运行时的垃圾回收实现了高效的内存自动管理，同时提供了 interface、slice、map 等内建数据类型，显著提升了开发效率和程序性能。然而，这些特性也带来与传统 C/C++语言不同的内存性能与安全性挑战。本文通过静态代码分析，实证研究了 Go 语言程序的内存性能与安全问题。通过设计基于 CodeQL 的 GitHub 开源代码分析框架 *PatStat*，利用声明式语言 QL 分析开源仓库中的内存性能相关的代码模式，并辅助人工总结和自动检测内存安全问题。研究涵盖 Go 程序的内存访问特征和安全问题模式。在分析 996 个近一年内更新的 Go 语言开源项目后发现：Go 程序中域访问和解引用操作在内存访问中占比较高，分别为 25.44%与 17.63%，意味着需要在程序分析或优化中关注域敏感性和指向分析。此外，Go 特有的 interface 类型转换引发的隐式内存分配也是程序优化的重点。通过人工分析 130 个涉及内存泄漏、无效内存地址或空指针解引用、悬垂指针的 issues，总结 10 类问题模式，其中悬垂指针问题在 Go 语言中较少见。这些问题通常需要约 30 天修复，但大多只需修改数十行代码即可完成。研究成果为优化 Go 语言程序和减缓内存安全问题提供了重要参考。此外，针对包含切片表达式赋值，可能导致内存泄漏的问题模式(L2-3)开发代码检查工具，并在真实项目中报告了 6 个问题，其中 1 个问题得到了肯定回复。

关键词: Go; 内存性能; 静态代码分析; 内存安全; 实证分析

中文引用格式: 李清伟,丁伯尧,张昱,陈金宝.Go 语言程序的内存性能与安全问题实证研究.软件学报,TODO

英文引用格式: Li QW, Ding BY, Zhang Y, Chen JB. Empirical Study of Memory Performance and Safety Issues of Go Programs. Ruan Jian Xue Bao/Journal of Software,TODO (in Chinese). **TODO**

Empirical Study of Memory Performance and Safety Issues of Go Programs

LI Qing-Wei[1], DING Bo-Yao[1], ZHANG Yu[1], CHEN Jin-Bao[1]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)

Abstract: The Go programming language(Golang), as an emerging programming language, achieves efficient automatic memory management through compile-time escape analysis and runtime garbage collection. It also offers built-in data types such as interface, slice, and map, significantly improving the development efficiency and program performance. However, these features also lead to memory performance and safety issues that distinct from those in traditional languages like C/C++. This paper uses static code analysis to empirically study memory performance and safety issues in Go programs. We design the framework *PatStat* to analyze open-source repositories in GitHub based on CodeQL. The framework employs the declarative language QL to analyze code patterns related to memory performance in open-source repositories, assisting manual summarization and automatic detection of memory safety issues. The research covers memory access characteristics and patterns of memory safety issues. The analysis of 996 Go open-source projects updated within the past year reveals that field accesses and dereference operations account for a significant proportion of memory accesses in Go programs, at 25.44% and 17.63%, respectively. This highlights the need to focus on field sensitivity and points-to analysis in Go program analysis and optimization. Additionally, implicit memory allocation caused by Go's unique interface type conversions is also key area for optimization. Through manual analysis of 130 issues involving memory leaks, invalid memory address or null pointer dereference, and dangling pointer, the study identifies 10 issue patterns, noting that dangling pointer issues are rare in Go. These issues take around 30 days to fix, but often require only a few dozen lines of code changes. These findings provide critical insights for optimizing Go programs and mitigating memory safety issues. Furthermore, a linter was developed for detecting the problematic pattern (L2-3) involving assignment of a slice expression, which could potentially lead to memory leaks. This tool identified 6 issues in real-world projects, one of which received positive acknowledgement.
Key words: Go;Memory Performance; Static Code Analysis; Memory Safety; Empirical Study

* 基金项目: 国家自然科学基金 (No.62272434)
收稿日期：2025-04-27；修改日期：2025-04-27; 采用时间: 2025-05-06

Go 语言是一门新兴现代编程语言，由 Google 在 2009 年开源。Go 语言凭借其语法特性简洁、内存自动管理、编译速度快、原生支持高并发等特点，在许多如容器、云原生、数据库、无服务器架构、微服务等新兴软件场景[1]中被广泛使用。

作为一门兼顾开发效率与执行效率的新兴语言，Go 语言在内存管理方面兼具灵活性与自动化。一方面，它支持类似 C/C++的指针和寻址操作，提供了较高的灵活性；另一方面，它通过引入垃圾回收进行自动内存管理，降低了人工内存管理的成本。Go 的内存管理采用了编译与运行时结合的方式。在编译时，利用逃逸分析确定对象的分配位置，将部分对象分配在栈上以提高程序的内存性能；在运行时，垃圾回收器负责动态内存的自动回收，同时通过检测机制在一定程度上避免诸如内存泄漏、空指针解引用等安全问题。然而，这套独特的内存访问与管理机制，也使得 Go 程序面临独特的内存安全和性能问题。

Go 语言的特殊内存管理机制虽然带来了便捷性，但也存在一定的性能问题和内存安全隐患，这些问题引发了社区和学术界的讨论和改进尝试。

一方面，Go 的垃圾回收机制在处理生命期相似的对象时效率有限。例如，Go 社区提案 Issue#51317 [2]建议引入 arena 机制，通过整体分配和释放生命期相近对象的内存区域来提升性能，该功能已在 Go 1.20 中实验性支持。此外，一些编程模式也可能引发性能问题。Wang 等[3]发现，将对象转换为接口并作为实参传递的操作可能导致不必要的堆分配，为此他们用正则表达式识别相应的模式并绕过逃逸分析来减少堆内存分配量。Ding 等[4]研究表明，Go 的逃逸分析算法存在保守建模问题，导致一些不必要的堆内存分配，他们通过为逃逸分析引入域敏感性和指向分析，能够在增加较小编译开销的情况下提升分析精度和程序性能。Peng 等[5]则通过显式识别和回收短生命期的堆对象，减少了垃圾回收的开销并提高了内存利用效率。

另一方面，尽管 Go 语言引入了垃圾回收和运行时检测机制，但却无法完全避免内存安全问题。例如，在远程过程调用框架 grpc-go[6] 中的 Issue#4758[7]中，在客户端连接数量减少、服务负载降低后，内存未能按预期减少，导致内存泄漏。此外，并发问题也是开发中常见的挑战。Liu 等[8] 使用 GCatch 工具发现了 149 个并发阻塞问题，而 Veileborg 等[9]进一步发现了 104 个并发阻塞问题。这些表明，尽管 Go 的内存管理机制已经相对完善，但仍有改进空间，开发者需要关注并应对相关问题。

针对上述内存性能与安全问题，静态代码模式识别与优化往往是有效的解决方案。例如，Selakovic 等[10]总结了 JavaScript 性能问题的常见代码模式，指导开发者和分析人员进行优化。Liu 等[8]利用 GCatch 工具识别信道阻塞问题的代码模式，并通过 GFix 自动修复。Chabbi 等[11]则针对数据竞争问题总结了相关的代码模式。Go 语言社区也存在较多通过规则和模式匹配的工具（如 Staticcheck[12], go vet[13], golangci- lint[14], gosec[15], revive[16]等），用于检测非内存安全问题和并发问题。然而，针对 Go 的内存性能优化和非并发类内存安全问题的模式识别和检测的工作仍较少。

基于以上观察，本文希望通过静态代码分析，对 Go 语言的内存性能优化问题和非并发类内存安全问题进行实证研究。研究将探索 Go 语言的内存访问特征及其问题模式，帮助开发者识别并避免潜在的问题，并为社区提供有关内存管理优化的见解，推动 Go 工具链的改进。

本文主要围绕以下三个研究问题（RQ）展开实证探究:

RQ1: Go 语言程序的内存访问特征是什么？这些访问特征如何影响内存问题的识别和优化？

在现代语言和编译器的设计中，为平衡编译速度和程序性能，分析和优化算法通常聚焦于最常见的场景，这要求对程序特征有整体的理解。本研究旨在探索现实世界中 Go 语言程序在其独特内存管理与访问机制下的内存访问特征，识别影响性能的内存使用模式，并据此为 Go 语言编译器或分析工具的改进提供指导。

RQ2: Go 语言程序存在哪些内存安全问题，这些问题在开源项目中的分布情况和常见代码模式如何？

为了减少内存安全问题的影响，并帮助开发者和静态分析工具研制者更有效地识别和解决这些问题，本研究旨在分析 Go 程序中常见内存安全问题（如内存泄漏、空指针解引用）的分布与表现，探索其主流的问题代码模式，同时评估这些问题在开源项目中的修复方式、所需成本和规模，以提供改进方向和优化建议。

RQ3: Go 语言程序中的内存安全问题模式如何归纳与检测？

通过归纳和总结内存问题模式，如果能够针对一些相对简单的模式实现自动检测，则有助于减少类似问题在社区中的重复出现。虽然 Go 语言生态中已有一些代码检查工具，这些工具在提升软件质量和可靠性方面发挥了重要作用，但它们可能未覆盖某些特定的内存安全问题模式。本研究以 RQ1 和 RQ2 的方向为基础，进一步探索如何利用静态代码模式实现内存问题的高效检测和定位。此外，本文还设计并实现了一种用于检测内存泄漏问题中特定模式（L2-3）的自动检测工具。

本文的主要贡献如下：

1. 设计一个开源项目分析框架 *PatStat*，它不仅能分析和统计 Go 项目中各类基本访存操作的分布，还支持对不限于 Go 语言的开源项目，通过关键词搜索 Issues 和 pull requests(PRs)，进行问题及其修复的统计分析，

并进一步辅助人工深入分析。

2. 用 *PatStat* 框架分析 996 个一年内有更新的 Go 语言开源项目，结果表明:

(a)在各类基本内存访问操作中，域访问和解引用操作的占比分别为 25.44%和 17.63%，仅次于直接变量访问操作(占比 51.10%)，在分析和优化时可重点关注。进一步的研究表明，对于如 p.f.a 这类连续域访问和解引用操作，只需对单层操作如 p.f 进行精确建模即可达到较为理想的分析精度。

(b)具体类型对象转换为接口的模式可能引发隐式堆分配。在所有模式中，赋值语句和调用语句占比最高，分别为 20.16%和 77.48%，合计 97.64%，为主要模式。进一步分析发现，这两类模式中分别有 23.65%和 37.24%的实例可能引起隐式堆分配，有潜在的内存优化空间，值得重点关注。

3.通过对内存相关的 GitHub Issues 的分析，总结出如下非并发内存安全问题在 Go 语言中的表现及模式：

(a)内存泄漏：尽管 Go 引入了垃圾回收机制，但在新兴应用中内存泄漏问题依然存在，且模式多样。例如，某些资源仍需手动释放，defer 延迟机制无法完全解决遗漏调用的问题(L1-2 模式)，slice 表达式处理不当也可能导致内存泄漏(L2-3 模式)等（见第 4.3.1 节）；

(b)空指针解引用：空指针解引用问题在 Go 语言中仍频繁发生，已识别出五类主要问题模式，适合通过代码检查工具进行进一步检测和预防（见第 4.3.2 节）；

(c)悬垂指针：虽然在 Go 的语言机制下几乎不会出现悬垂指针问题，但当开发者自行设计引用机制，则仍可能引发此类问题（见第 4.3.3 节）。此类问题的检查需要更高层次的抽象分析，而非局限于语句级或函数级的简单分析。

4.在 RQ2 的研究中，本文发现并总结了 Go 代码中可能导致内存泄漏的 L2-3 模式，并据此开发了代码检查工具。通过基于 *PatStat* 对 996 个项目生成的 966 个项目数据库进行分析，得到如下结果：

(a)在 966 个项目中，共报告了 574 处可能存在问题的代码位置。

(b)对其中 70 个报告进行人工审查，发现 6 个为正确报告，64 个为误报。将人工审查正确的报告形成了 5 个 Issues(其中两个问题合并为一个 Issue)提交至 GitHub，最终获得 1 个肯定的回复，2 个认为是误报，还有 2 个尚为收到回复。

本文按如下结构进行组织：第 1 节介绍 Go 语言相关的背景知识以及研究动机，第 2 节介绍所设计的 *PatStat* 分析框架；第 3-5 节分别介绍对三个 RQ 的研究与分析结果；第 6 节讨论了研究方法的有效范围与局限性，第 7 节介绍相关工作。

1 背景与动机

本节结合现实世界中的内存性能及安全问题，阐述 Go 语言相关背景以及本文的研究动机。

1.1 Go 语言内存管理方式

Go 语言的内存管理结合了逃逸分析[17,18]和垃圾回收机制[19]。通过基于标记-清除的垃圾回收器，Go 自动回收不再使用的内存，简化了手动内存管理的复杂性。同时，编译器通过逃逸分析决定对象的分配位置：栈上用于存储短生命期的局部变量，堆上则存储需要跨越函数边界的对象。虽然这套机制有效减少双重释放、悬垂指针等问题，降低了内存泄漏的风险，但开发者仍需留意持久引用可能引发的慢泄漏问题[20]。

此外，为了提升开发效率，Go 语言提供了如接口 interface,切片 slice,映射 map 等内建数据类型。interface 类型的变量可以存储任意定义了特定方法的类型实例[21]。slice 是对底层数组一段连续内存区域的描述，可通过下标访问元素[22]。map 则是一组无序元素对的集合，通过键进行索引操作[23]。

1.2 Go 语言程序中的内存性能问题

Go 语言的内存性能问题主要在于过多的堆对象会增加运行时垃圾收集的开销，从而影响程序性能。现实中的解决方案主要包括两方面：1. 优化运行时内存管理机制；2. 减少堆内存的分配与使用。

在优化内存管理机制方面，分布式数据库 cockroach[24]发现在特定模式下创建的字符串有着类似的生命期，因此引入自行设计的 arena 区域管理机制，对这些字符串进行统一分配与释放，显著提升内存性能。

在减少堆内存的分配与使用方面，Go编译器的逃逸分析通过对程序中的表达式和语句进行建模来确定对象的生命期。对于不会被作用域外引用的短生命期对象，逃逸分析将其分配在栈上以减轻垃圾回收器的负担。Wang 等[3]通过识别那些被转换为接口并作为实参传递的堆分配对象模式，再使用uintptr绕过逃逸分析，将堆对象改为栈分配，减少了堆内存的使用。Marr[25] 通过调整&符号的位置，在保持相同功能的前提下，减少了不必要的堆内存分配。这些研究表明，基于特定静态代码模式的优化是提升内存性能的有效手段。

另外，Go 编译器的逃逸分析在处理域访问和解引用操作时采用了保守建模，导致性能优化空间受限。例如，

在 Go 语言官方测例中（代码 1 和代码 2），由于建模不够精确，程序中原本可以分配在栈上的对象 i 被判定为逃逸到堆上，增加了不必要的堆内存分配，降低了内存性能。鉴于此，本文对内存访问相关的静态代码模式进行了分析与总结，旨在为 Go 编译器与静态分析工具的开发者提供参考。

代码 1 escape_indir.go：解引用保守建模导致 i 堆分配

```
1.  type ConstPtr struct{ p *int }
2.  func constptr0() {
3.    i := 0
4.    x := &ConstPtr{}
5.    // BAD: i should not escape here
6.    x.p = &i
7.    _ = x
8.  }
```

代码 2 escape_field.go：域不敏感建模导致 i 堆分配

```
1.  var sink interface{}
2.  func field1() {
3.    i := 0
4.    var x X
5.    // BAD: &i should not escape
6.    x.p1 = &i
7.    sink = x.p2
8.  }
```

1.3 Go 语言程序中的内存安全问题

表 1 经典内存安全问题与 Go 语言采取的措施

经典内存安全问题	Go 语言采取的措施
内存泄漏（memory leak）	垃圾回收
释放后使用（use-after-free）	垃圾回收
双重释放（double-free）	垃圾回收
悬垂指针（dangling pointer）	逃逸分析与垃圾回收
下标越界（index out of bound）	运行时检查
空指针解引用（null(nil) pointer dereference）	运行时检查

传统的内存安全问题主要包括如表 1 所示的六大类。表中相应地总结了 Go 语言针对这些内存安全问题所采取的措施。Go 语言对这些内存安全问题采用两种处理方式，一类是运行时检测与错误报告，另一类是编译期分析与运行时机制结合。例如，对于内存泄漏、释放后使用、双重释放等问题，Go 语言采用垃圾回收的方式避免程序员手动管理内存带来的问题。针对需要显式释放的资源（如网络连接、数据库连接、文件描述符、锁等），Go 语言引入了 defer 关键字，便于开发者释放和清理资源。针对悬垂指针问题，Go 语言采用逃逸分析判断对象的生命期，将生命期较长的对象分配在堆上，并通过垃圾回收避免悬垂指针。针对数组访问越界和空指针解引用问题，Go 编译器生成检测代码并在运行时进行检查。

尽管 Go 语言在内存安全方面做出了诸多努力，本文的实证分析表明，内存泄漏和空指针解引用这两类问题仍然在 Go 语言开源项目中存在，表明当前的机制尚未完全解决这些问题。

虽然 Go 语言引入垃圾回收(GC)来减少内存管理的复杂性，但这并不意味着能够完全避免内存泄漏。Go 的 GC 使用基于标记-清扫的算法，在运行时会从根集出发，追踪和识别对象的引用关系，将不再被使用的对象回收。然而，由于 GC 需要追踪不同对象之间的引用关系，即使部分对象已经不再被访问，只要这些对象仍然被其他对象引用，它们就无法被回收，从而造成内存泄漏问题。另外，Go 语言通过 defer 关键字来帮助开发者管理资源释放，但仍然需要程序员显式地调用释放函数。如果开发者未能正确在适当时机使用 defer 或忘记释放资源，那么这些资源就会一直占用内存，从而可能导致内存泄漏问题。

例如，grpc仓库曾报告了如图 1所示的内存泄漏问题issue#4758[7]： 随着右图用户Clients连接数的不断增长，内存使用量不断增加。但是当用户退出连接之后，内存使用量仍然保持不变，没有被回收。导致该问题的代码如代码 3中被删减的代码，`q.queue[0]`所引用的对象后续不再被使用，但由于没有将`q.queue[0]`置为`nil`，导致垃圾回收器认为`q.queue[0]`所引用的对象仍然活跃，不能被回收。在该语句被执行多次后，就会存在较多对象无法被及时回收，造成内存泄漏问题。这类问题可以抽象总结为模式：元素类型为指针类型的切片在切片表达式操作前没有进行置空操作。由于这类模式容易为静态分析工具所识别，因此，希望总结这类简单问题模式，即回答RQ2，以便后续静态分析工具能够识别和检测这类问题。

另外，尽管 Go 语言通过生成代码插入运行时检查，防止数组越界和空指针解引用等问题，但是这些问题仍然会导致程序在运行时崩溃，从而终止执行以确保安全性。如果能使用静态分析工具在程序运行前发现这些问题，可以避免因测试覆盖不足导致问题在应用部署后才出现。静态分析也能够全面检查程序代码，覆盖所有可能的执行路径，从而更有效地发现潜在问题。因此，静态分析检测内存安全问题有重要意义，能够提高程序的稳定性和

可靠性。

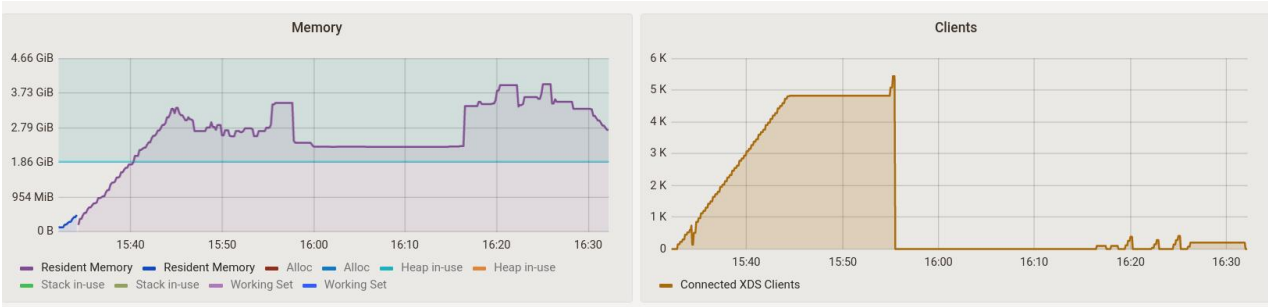

图 1 grpc/grpc-go/issues/4758 内存泄漏问题，在用户数量下降之后，内存使用量仍然保持不变

代码 3 grpc-issues#4758：slice 操作前未置空导致内存泄漏

```
1.  var item *itemType
2.  - item, q.queue = q.queue[0], q.queue[1:] //
    q.queue[0] will not be used forever
3.  + item = q.queue[0]
4.  + q.queue[0] = nil
5.  + q.queue = q.queue[1:]
6.  ... // destroy item
```

1.4 面向 Go 语言程序的内存安全问题检测工具

为了缓解内存安全问题，除了 Go 语言采用的运行时崩溃等机制，一些代码检查工具也被开发和使用。Staticcheck[12]是一个较为先进的 Go 语言代码检查工具，它能够检测标准库的误用、并发问题、冗余代码和内存性能等问题。针对内存安全问题，Staticcheck 包括了对冗余 nil 检查[26–29]、可能发生空指针解引用的检测[30]等分析规则。golangci-lint[14]集成了多个代码检查工具工具，支持代码风格检查，未使用函数和变量的检查等。go vet 工具[13] 作为 Go 语言内置的检查工具，可以检测如 defer 使用时的常见错误、不必要的比较函数指针和 nil 模式等。

尽管这些检测工具在发现和修复常见的编程错误方面发挥了重要作用，但它们仍然存在一定的局限性。因此，对 Go 语言内存安全问题进行实证分析显得尤为重要。通过对 Go 代码中的内存安全问题进行深入研究，可以帮助开发者设计和构建更有效的代码检查工具，从而更早发现潜在的内存安全隐患。

2 方法设计

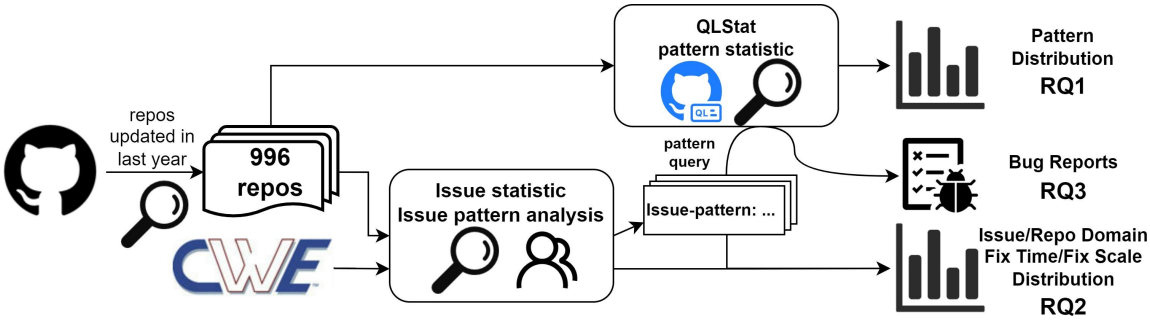

图 2 *PatStat* 框架整体设计

为了深入探讨 Go 语言的内存性能和安全问题及其关联模式，本文设计了如图 2 所示的分析框架 *PatStat*。针对 RQ1，基于CodeQL[31]设计了统计分析工具 *QLStat*，能对不同基本访存操作进行全面统计分析。针对RQ2，设计了 *IssueAnalyzer* 工具，能通过关键词检索 GitHub 的Issues/pull request(PRs)，结合统计与人工分析，总结 Go 语

言中内存安全问题的分布特征及常见代码模式。针对 RQ3，为RQ2 中识别的典型内存问题模式编写查询语句，利用 *QLStat* 工具对选择的 Go 代码仓库进行扫描查询，以发现潜在的内存安全问题，并探索可能的自动检测方法。

本文通过 GitHub 的 REST API 获取了最近一年内有更新的 Go 语言代码仓库。选择最近一年内更新的仓库，旨在确保其处于积极维护状态，能够反映 Go 语言生态的活跃部分及当前发展趋势和实际应用情况。最终，共收集到 996 个仓库，并且基于这些仓库开展后续的模式统计分析。

2.1 RQ1：访存操作的模式统计

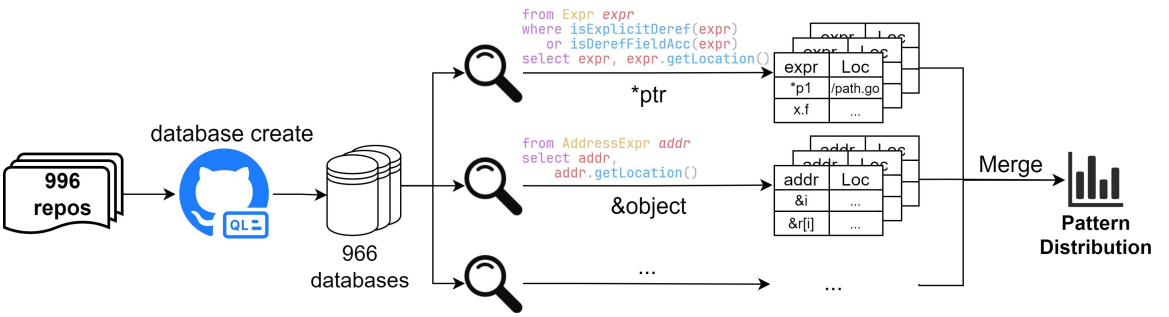

图 3 访存操作的模式统计工具 *QLStat*

为了回答RQ1，设计了如图 3所示的*QLStat*，基于抽象语法树(AST)识别Go语言项目中不同基本访存操作并统计分布情况，为编译器优化提供指导。*QLStat*基 于 CodeQL对代码仓库进行模式匹配，合并各仓库的分析结果，从而实现不同基本访存操作的统计分析。CodeQL是一款代码扫描工具，能够提取程序的多种信息，如抽象语法树(AST)、控制流图(CFG)、数据流图(DFG)、调用图(CG)以及类型信息，并将其存储到关系数据库中。通过数据库查询语言QL[32]，可以编写匹配特定代码模式的查询语句，从数据库中查询相关模式实例及其信息。借助CodeQL，*QLStat*工具能够支持包括Go语言在内的多种语言的批量仓库的模式统计分析，为编译器优化提供数据支撑和决策依据。

QLStat 通过以下步骤来对代码仓库中的访存操作模式进行匹配与分析：

- **生成数据库.** *QLStat* 首先对收集的 996 个代码仓库进行分析，成功生成了 966 个数据库。
- **基于数据库统计访存操作模式.** 在生成数据库后，*QLStat* 基于 AST 使用模式匹配查询语句，统计不同基本访存操作的实例分布。例如针对取地址访存操作(&object)，*QLStat* 的查询语句是相应的 from AddressExpr ... select ...语句。在某个仓库的查询结果出现了&i 和&r[i]两个实例，表明在该代码仓库中存在两个取地址访存操作实例。
- **结果合并与分析.** *QLStat* 将所有数据库的查询结果合并，生成不同基本访存操作的全局分布数据。这些分布数据为编译器优化提供了重要参考，帮助开发者识别程序中频繁的访存操作，从而制定针对性的优化策略，提高程序性能。

2.2 RQ2：内存安全问题分析

为了回答 RQ2，设计了如图 4 所示的 *IssueAnalyzer* 工具，以帮助分析常见内存问题在 Go 语言中的存在性、分布、项目领域分布以及内存问题的模式特征。

Issues/PRs 获取与分析 *IssueAnalyzer* 首先对 996 个仓库进行搜索，获取过去 5 年内更新的 Issues/PRs。选择较新的 Issues/PRs 是为了更准确地反映当前 Go 语言项目中的实际问题。在搜索过程中，*IssueAnalyzer* 聚焦于标题、正文和评论中包含特定关键词的内容，从而构建和关键词相关的 Issues/PRs 集合。

在获得 Issues/PRs 的集合之后，*IssueAnalyzer* 从宏观和微观两个层面进行分析。宏观上，*IssueAnalyzer* 对不同关键词搜索到的 Issues/PRs 在各个仓库中的分布进行了统计分析，探讨哪些内存问题在 Go 语言项目中更为常见，更易发生。另外，*IssueAnalyzer* 还分析包含这些 Issues/PRs 的项目的领域，了解哪些领域的项目更容易受到内存问题的影响。微观上，*IssueAnalyzer* 分别对 Issues/PRs 采样并获得匹配的 Issues/PRs 对，人工总结常见的内存问题模式，并基于配对的 Issues/PRs 对，统计修复时间和代码变更规模。依据分析结果，为开发者和静态分析工具开发者提供建议，帮助识别和解决内存问题。

Issue 和 PR 的匹配 为了获得更加准确的结论，作者对图 4 中采样得到的 Issues 和 PRs 进行了人工配对。具体来

说，对于一条 Issue，人工查找提及这些 Issue 的 PR；对于一条 PR，查找 PR 可能修复的 Issue。在完成 Issue 和 PR 的配对后，判断这些 PR 是否被成功合并。如果成功合并，则认为该 PR 修复了对应的 Issue，并将其作为最终的 Issue 和 PR 对保留下来。

修复记录统计规则 在完成 Issue 和 PR 的配对分析后，*IssueAnalyzer* 统计了 PRs 的修复时间和代码变更规模，以评估 PRs 涉及的内存问题需要多长时间、多少代码变更才能被修复，从而回答这些内存问题是否容易被修复。

本文采用的修复时间间隔定义为：当一个 PR 与某个 Issue 匹配时，以 Issue 的创建时间作为修复的开始时间，否则以 PR 的创建时间作为开始时间；当一个 PR 被关闭时，以 PR 的关闭时间作为修复结束时间，若 PR 仍处于 open 状态时，则以本文获取到该 PR 的时间为修复结束时间。修复的时间间隔为修复结束时间减去修复开始时间。这样的定义方式可以尽可能准确地反映一个问题从发现到最终修复所花费的时间。

为了评估代码变更的复杂性，*IssueAnalyzer* 将一个 PR 中所涉及的代码增加与删除行数的和作为代码变更规模。

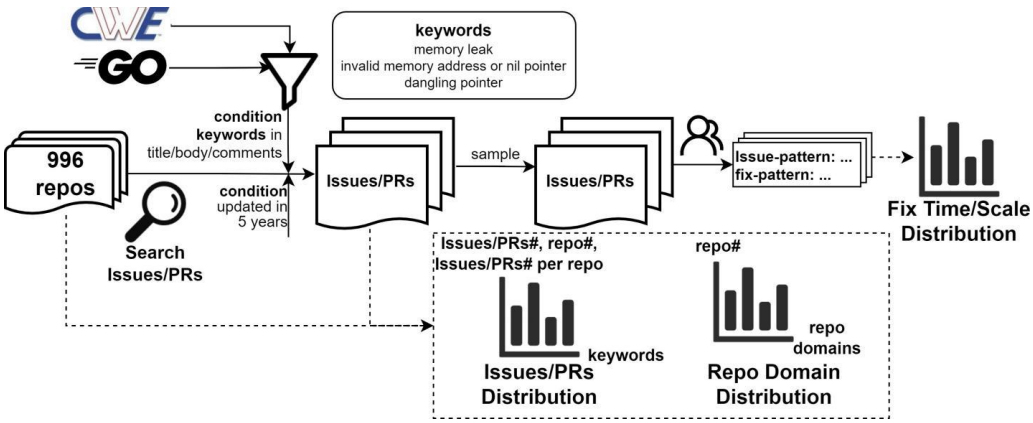

图 4 内存安全问题分析工具 *IssueAnalyzer*

2.3 RQ3：L2-3 问题模式的检测

基于RQ2 中归纳的内存安全问题模式，本文探索可能的自动检测方法。以 L2-3 模式为例，本文编写了相应的检测语句，使用 *QLStat* 对966 个项目数据库进行扫描分析。检测结果展示了 L2-3 模式在真实 Go 语言项目中的分布情况。最终，本文将检测到的问题反馈给相关项目，以协助开发者改进代码质量和内存安全性。

3 RQ1: Go 语言程序的基本访存特征

基本访存操作的分布对编译器优化和性能分析的建模具有重要意义。本文使用 *QLStat* 工具在程序的抽象语法树上，基于 966 个Go 语言项目数据库进行了静态代码特征的分析，主要包括以下两个方面：

1) 基本访存模式分布（第 3.1 节）：揭示 Go 语言中的基本访存模式及它们在实际程序中的分布情况，统计各类访存操作的频率。

2) 接口类型与隐式内存分配（第 3.2 节）：接口类型的使用可能引发隐式内存分配，造成潜在的内存性能问题。本部分深入分析接口类型转换模式及其对程序性能的影响。

3.1 基本访存操作的分布

基本访存操作分布 根 据 Go语言官方提供的规范[33]，本文将Go语言程序中的基本内存访问操作总结为7类，如表2所示。其中，类别Direct为对变量的访问操作表达式，Deref为解引用表达式，Ref为取地址表达式，FieldAcc为域访问表达式，SliceIdx、ArrayIdx、MapIdx为索引模式，分别为对切片、数组和映射的索引。由于对变量的直接访问 (Direct) 广泛嵌套于其他表达式中，统计时会存在重复计数。为避免这一问题，*QLStat*对Direct模式的统计做了去重处理：当 Direct模式作为外层模式（如Ref, Deref等）的一部分时，不计入Direct模式。例如，在表达式&a中，对变量a的访问不计入Direct模式中。

对 966 个数据库的统计分析表明，除直接访问(Direct)之外，解引用(Deref)和域访问操作(FieldAcc)操作占比

最高，分别为 17.63%和 25.44%。这表明在编译器优化和静态分析工具的建模中，应重点关注这两类操作，以更有效地提高分析精度和优化性能。

表 2 基本访存操作的分布

形式	类别	数量	比例
a	Direct	17649448	51.10%
*a	Deref	6089052	17.63%
&a	Ref	1181230	3.42%
a.f	FieldAcc	8785663	25.44%
slice[i]	SliceIdx	511916	1.48%
array[i]	ArrayIdx	40872	0.12%
m[i]	MapIdx	283304	0.82%
总计		34541485	100.00%

表 3 连续域访问次数分布

连续域访问次数	数量	比例
1	18529964	93.32%
2	1122460	5.65%
⩾ 3	204614	1.03%
总计	19857038	100.00%

表 4 连续解引用次数分布

连续解引用次数	数量	比例
1	305679	97.21%
2	8765	2.79%
总计	314444	100.00%

在程序分析算法设计中，为了提升算法分析精度，通常需要使用更细粒度的建模方式，获取更多的分析事实。但是这也会导致算法开销显著增加，因此如何平衡使用更高精度分析算法的收益与代价是所有设计者都要面临的问题。为了给 Go 程序的分析与语言优化提供更细粒度、更精确的参考，本文聚焦于统计结果中占比较高的解引用操作(Deref)和域访问操作(FieldAcc)，对连续域访问操作和连续解引用操作进行深入分析，以探索优化潜力。

连续域访问次数 连续域访问是指一个域访问表达式的基表达式仍然为域访问表达式，如 `a.f1.g2` 的基表达式 `a.f1` 仍是域访问表达式。连续域访问模式会影响域敏感分析的算法效率和空间开销。如果域敏感分析对所有程序中出现的域均进行建模，那么会有较大的空间浪费——部分域上的状态信息可能并不会被使用，但是仍然需要被更新。这也会导致算法执行效率较低。

例如，假设a结构体变量包含2个域f1和f2， 这两个域是包含两个域g1和g2的结构体。对于`y = a.f1.g1`语句，在域不敏感分析中，语句会被建模为`y = a`，这会导致分析不精确。而在域敏感分析中，f1和g1都需要被建模，分析更加精确，g1和g2的状态会被区分，但会带来较大的分析时空开销。如果将分析的粒度控制在只建模f1，而不区分g1和g2，即将语句建模为`y = a.f1`，则可以降低分析开销，同时保留f1和f2之间的区分。如果可以知道连续域访问次数的分布情况，就可以调整域敏感分析的粒度，从而在不损失太多精度的情况下提升算法执行效率和减少空间开销。

*QLStat*通过递归遍历所有域访问表达式，检查其基表达式是否仍为域访问表达式，从而计算连续域访问次数。统计结果如表 3所示。在所有域访问操作中，单次域访问占比约为93.32%；连续域访问次数为2的占比为5.65%，次数大于等于3的占比仅为1.03%。上述结果显示，在设计针对Go语言的域敏感程序分析算法时，关注单次域访问操作即可覆盖绝大多数场景，获得显著的精度收益。而对连续多次域访问操作的敏感性分析虽然可以提高精度，但其占比较小且处理难度较高，可能带来不成比例的时空代价，因此可以根据需求权衡是否支持。

连续解引用次数 连续解引用是指一个解引用表达式的子表达式仍然为解引用表达式，例如`**p`的子表达式`*p`仍是解引用表达式。如果需要精确知道`**p`所指向的对象，需要进行两次指向集合计算，这会带来较大的时间和空间开销。但是部分指向分析下游应用（如逃逸分析）并不需要如此高的精度。如果可以知道连续解引用次数的分布，那么下游应用就可以减少指向集合计算次数，在保证效率的同时达到相对理想的分析效果。

*QLStat*通过递归方式遍历所有解引用表达式，计算连续解引用次数。统计结果如表 4所示，与连续域访问次数的分布类似，单次解引用操作占绝大多数，为97.21%，连续解引用次数为2的占了2.79%。上述结果表明，在Go语言中使用指针分析结果时，可以对连续解引用次数大于等于2的场景采用保守建模策略。这种方式在显著降低分析开销的同时，依然能够提供较高的精度，满足实际需求。

结论 1 - RQ1

在基本访存操作中，域访问操作和解引用操作占有较高的比例，分别为 25.44%和 17.63%，在编译器优化和静态分析工具设计应重点关注。此外，单次域访问占比超过 93%，单次解引用操作占比超过 97%。因此在设计域敏感分析和使用指针分析结果时，仅需要考虑单次域访问和单次解引用即可覆盖绝大部分场景。这种设计能显著降低分析开销，同时实现快速且高精度的分析目标。

3.2 接口类型的分析

接口类型变量在 Go 的实现中通过一个结构体表示，如代码 4 所示，包含两个域，其中一个域是类型方法表指针 tab，用于描述指向类型描述符或者方法表，另外一个域是数据指针 data，用于指向实际数据对象。当将非接口类型的值赋给接口类型变量时，通常需要在堆上分配一块新的空间，并让 data 指向该新空间以表示新对象。只有当非接口类型是指针时，接口变量的 data 才直接指向原始对象，而无需额外的堆分配。因此，接口类型赋值可能导致隐式内存分配，从而引发潜在的内存性能问题。

由于接口类型转换可能导致隐式内存分配，并对 内存性能与程序分析建模产生影响，*QLStat*根据 Go语言官方规范[33]，在抽象语法树层面总结了5类可能引发新分配对象的接口操作模式，如表 5所示。在这 5类模式中，assign模式中 LHS为接口类型，call模式中形参类型均为接口类型，conversion模式为显式接口转换，ret模式中返回类型为接口类型，send模式中发送到通道的数据类型为接口类型。除conversion模式为显式转换外，其他模式均存在到接口类型的隐式转换。统计结果显示，assign和call模式占据所有模式中的绝大部分，分别为20.16%和77.48%，表明分析和优化与接口类型相关的代码时，需要重点关注这两类模式。具体来说，assign模式对函数内的动态内存分配分析与优化提出挑战，而call模式则针对跨函数的过程间动态内存分配分析与优化提出挑战。另外ret模式也存在一定比例，揭示了过程间动态内存分配优化的潜力。

代码 4 接口类型实现与接口类型转换

```
1.  // underlying implementation of non-empty interface
2.  type iface struct {
3.    tab  *itab // type descriptor or table pointer
4.    data unsafe.Pointer // data pointer
5.  }
6.  var a int
7.  var i interface{} = a // implicit heap allocation
```

表 5 转换为接口的模式分布以及被转换类型分布

形式	类别	数量	比例	接口类型	非指针类型	指针类型
LHS = RHS	assign	920949	20.16%	71.69%	23.65%	4.66%
o.m(a1, a2)	call	3540073	77.48%	45.51%	37.24%	17.26%
(interface{})(src)	conversion	4751	0.10%	5.20%	64.11%	30.69%
return a	ret	100806	2.21%	6.56%	56.04%	37.08%
ch <- a	send	2218	0.05%	9.20%	81.61%	9.20%
总计		4568797	100.00%	49.92%	34.96%	15.11%

进一步分析被转换为接口类型的数据类型的分布，本文将这些类型分为3类：接口类型，非指针类型和指针类型。接口类型和指针类型在转化为接口类型时不需要动态分配新对象。非指针类型转换为接口类型需要动态分配新的转换前类型对象。三种类型对象被转换为接口类型的分布如表 5所示。在assign和call模式中，接口类型的转换频率较高，在conversion, ret和send模式中，非指针类型占据主导，这导致了显著的隐式堆内存分配需求。因此，在进行内存优化时，应优先关注非指针类型转换的分配问题，以减少隐式分配带来的性能开销。

结论 2 - RQ1

在接口类型转换的模式中，assign 模式和call 模式两类模式占据绝大部分，表明分析接口相关的内存分配时应重点关注这两类模式。此外，由于非指针类型转换为接口类型需要在堆上新分配对象，在涉及优化 conversion, ret, send 模式时需要特别注意这些隐式的堆内存分配。由于 assign 和call 模式基数较大，其中非指针类型转换为接口类型引发的隐式堆分配也是值得内存优化关注的点。

4 RQ2: 内存安全问题模式

围绕开源 Go 语言项目中的内存安全问题，本节首先确定研究内存安全问题的关键词（见第 4.1 节），然后统计通过不同关键词搜索得到的 Issues 数量分布，以回答 RQ2 中的内存安全问题存在性和分布问题，并分析了出现较多内存安全问题的仓库的领域分布，以揭示 RQ2 中仓库领域与内存安全问题之间的关系（见第 4.2 节）。最后本节总结了与 3 个关键词相关的内存安全问题的模式（见第 4.3.1-4.3.3 节）。

4.1 内存安全相关问题筛选

本文从CVE[34], CWE[35]和Go官方维护的漏洞数据库[36]搜索内存安全问题。其中，只有CWE 提供了对内存安全问题的分类总结，包含了C、C++、Java、PHP等语言中的常见内存安全问题以及不同年份的Top N安全问题视图[37]。由于CWE中未涵盖Go语言特定内存问题的视图，本文基于CWE的两个视图——CWE Top 25(2022)[38]和Software Written in C++[39]，筛除非内存安全问题和在Go语言中不太可能出现的问题，结合Go语言在运行时检查中常见的报错信息[40]，筛选出以下3个关键词，分别为“memory leak”, “invalid memory address OR nil pointer” 和“dangling pointer”。其中“memory leak” 是指内存泄漏问题，简记为 LEAK；“invalid memory address OR nil pointer”通常在应用程序执行时由Go运行时检测到空指针解引用时报错，简记为NPD。“dangling pointer”则是指资源对象（如内存对象或文件描述符对象）被释放之后，内存中还包含对这些已释放资源的引用，简记为DP。基于这3个关键词，本文使用图4所展示的方法，搜索了996个Go语言仓库过去5年内更新的Issues和pull requests(PRs)，以及包含memory关键词的Issues/PRs，后者可视为内存安全问题的总集。基于这些Issues/PRs，对含内存安全问题的仓库的领域特点进行分析，藉此进一步探索内存安全问题在不同领域的发生情况。

4.2 内存安全问题分布

本小节将基于内存安全相关的 Issues/PRs 的数量，分析内存安全问题在实际 Go 应用中的分布情况。

4.2.1 内存安全问题数量分布

根据图4，针对从GitHub中获取的996个最近1年内更新的仓库，通过设置3类关键词进行搜索，得到了不同关键词的Issues/PRs数量分布如表 6所示。从数量上看，包含LEAK和 NPD的 Issues/PRs数量较多，分别为7126、13007条；这两个关键词涉及的仓库数量也较多，分别为573和 708个。表 6中 avg列显示每个仓库内包含的Issues/PRs数量的平均值，min到 max列给出以25%为间隔的分位数。数据显示，75%的仓库至多只有2个可能为悬垂指针问题(DP)。而同样比例的仓库中，至多有10条内存泄漏问题(LEAK)和 16条空指针解引用问题(NPD)，表明内存泄漏和空指针解引用问题更为常见。因此，可以认为内存泄漏和空指针解引用问题在Go语言中仍然存在，而悬垂指针问题在 Go语言中较少。

4.2.2 内存安全问题与代码仓库所属领域的关系

为理解内存安全问题与代码仓库所属领域之间的关系，本文进一步收集通过memory关键词搜索得到的83850条 Issues/PRs关联的仓库所在的领域。在996个仓库中，有185个仓库的 Issues/PRs数量占总量的80%，剩余的大部分仓库只占不足20%。因此，本文选取这185个仓库作为更具有代表性的仓库来进行分析。

在对仓库进行领域分析时，本文使用了Costa等[41]提供的标注数据集，该数据集用于反映使用unsafe package的仓库领域信息。对于不在该数据集中出现的仓库，本文根据数据集中的领域标签进行标注。仓库领域的统计结果如表7所示。结果显示：开发工具、云/微服务、容器/虚拟机等领域的仓库存在相对较多的内存相关问题。其中开发工具类仓库占比最高，这可能与Go在CI/CD工具、CLI命令行工具等开发场景中的广泛应用有关。而对于云/微服务、容器/虚拟机两种类型的仓库，由于它们通常需要占用更多的内存资源，内存相关问题也更为显著。

表 6 搜索不同关键词得到的 Issues/PRs 数量与分布信息

avg-max 列以每个仓库所包含的 Issues 数量为分析单元。avg 列为平均值。min-max 列为分位数。

关键词	Issues/PRs 数量	仓库数量	avg	min	25%	50%	75%	max
memory leak (LEAK)	7126	573	12.4	1	2	4	10	540
invalid memory address OR nil pointer (NPD)	13007	708	18.4	1	2	6	16	1000
dangling pointer (DP)	203	91	2.2	1	1	1	2	19

由于GitHub API的限制，每次搜索最多只能返回1000条Issues/PRs[42]。这意味着通过memory关键词搜索得到的结果并非全集，会影响上述结论的有效性。然而，进一步分析显示，只有17个仓库的Issues/PRs数量达到了上限，这些仓库更能代表内存安全问题在Go语言中的现状。表8显示了这17个仓库的领域分析结果，它们更多出现

在开发工具、容器/虚拟机、数据库存储等领域，和上述代表性仓库的领域分布结论基本保持一致。因此，开发工具、云/微服务、容器/虚拟机和数据库存储类应用更有可能发生Go语言内存安全问题。

表 7 内存安全问题代表性仓库领域统计

仓库领域	仓库数量	占比 (%)
Development Tools	46	24.86
Cloud/Microservices	38	20.54
Container/VM	21	11.35
Database/Storage	17	9.19
Networking/ Messaging	15	8.11
Monitoring	13	7.03
Security	9	4.86
OS/File System	5	2.7
Web	5	2.7
Blockchain	4	2.16
Misc	3	1.62
Programming Language Support	3	1.62
UI	3	1.62
Compiler	1	0.54
ML/Scientific Library	1	0.54
Multimedia	1	0.54
总计	185	

表 8 含 memory 的 Issues/PRs 数达搜索上限的仓库及领域

仓库名称	仓库 Issues 数量	领域
cockroach	94936	Database/Storage
sourcegraph	58795	Development Tools
kubernetes	53719	Container/VM
tidb	44191	Database/Storage
istio	39286	Cloud/Microservices
terraform	34479	Development Tools
milvus	30354	Database/Storage
beats	28684	Cloud/Microservices
rancher	26911	Development Tools
cilium	26402	Networking/Messaging
podman	21016	Development Tools
nomad	20218	Development Tools
minikube	15084	Development Tools
moby	12510	Container/VM
go-ethereum	10512	Blockchain
kubevirt	9722	Container/VM
compose	5962	Container/VM

结论 3 - RQ2

内存泄漏、无效内存地址或空指针解引用两类内存安全问题在 Go 语言中仍然广泛存在，在 996 个仓库中，每个仓库平均包含 12.4 条内存泄漏, 18.4 条无效内存地址或空指针解引用的 Issues/PRs，中位数分别为 4 和6 条。相比之下，悬垂指针问题在 Go 语言中出现的频率较低。此外，开发工具、云/微服务、容器/虚拟机以及数据库存储类应用更容易出现 Go 语言中的内存安全问题。

4.3 内存问题的代码模式

为总结内存问题在代码中呈现的常见代码模式，进一步针对三类关键词分别人工分析了 50、30、50 条 Issues，将其归类为不同的问题类型，分析结果概览见表 9。在下面三小节中，本文将结合实际代码示例，详细介绍每类内存安全问题的代码模式，包含引发问题的代码模式及其相应的修复策略。

4.3.1 内存泄漏问题模式

在 7126 个内存泄漏Issues/PRs中，*IssueAnalyzer*采样了 50个Issues/PRs 进行分析。在这 50个Issues/PRs 中留下 23 个确定为内存泄漏问题的Issues/PRs进行了问题模式的人工分析。在分析过程中，本文结合Issue中的评论以及对应修复PRs中的修复代码，总结问题模式，将这些模式分为如表 9所示的L1-L6这 6 类模式。各问题模式的子类以及分布情况如表 10所示。

L1. 资源未被显式释放 在程序中缺少对资源显式释放函数的调用，从而造成资源的泄漏。在该问题类中包含 3 个小类，模式代码如代码 5所示。

L1-1: 配对资源未被一起释放。某些资源在定义时会引用其他配对资源。当申请特定资源时，通常会同时申请配对资源；而当特定资源释放时，配对资源也应一并释放。然而由于Go语言中自动垃圾回收并不会调用配对资源的释放函数，导致配对资源未被释放，进而产生内存泄漏。代码 5中1-6行展示了tidb的Issue#27353 [43]的简

化版问题。具体来说，Resource结构体的Close函数没有同时关闭配对的pairResource资源，导致配对资源无法被回收，从而产生内存泄漏。

L1-2: 资源使用后因缺失对其释放函数的调用导致资源无法被正确回收。代码 5中第 7-9行展示了grpc-go的Issue#6642 [44]的代码简化。在创建并使用`ticker`进行计时之后，缺少对其释放函数`Stop`的延迟调用，导致 ticker 资源无法被回收。仓库的贡献者评论指出，此类问题并非首次出现，而会长期不断发生[45]。除了因忘记延迟释放资源而导致的内存泄漏问题外，还存在在错误处理时缺失资源释放操作的情况，即代码 5中第 10-14行，该部分展示了tidb的 Issue#34666 [46]的简化版。在错误/异常处理分支，遗漏对已分配资源的释放，导致在异常发生时产生内存泄漏。另一种情况如代码5第 15-20行的L1-2-otherref模式，简化自btcd的Issue#2105 [47]，此时资源对象在不同条件下被不同对象引用，某个特定条件下遗漏了通过相应引用对象释放对资源的引用。在该问题中，布尔型变量`c.batch`指示资源`element`是被`c.batchList`还是被`c.requestList`引用的。修复前的代码未考虑当`c.batch`为 true 时，需要从`c.batchList`中删除对资源`element`的引用，潜在引起资源泄漏。

表 9 不同关键词人工分析的 Issues 数量

关键词	问题类	Issues 数量
内存泄漏 (LEAK)	L1. 资源未被显式释放	8
	L2. 不正确的显式释放	5
	L3. 并发	2
	L4. 依赖项内存泄漏	5
	L5. 拷贝大内存	1
	L6. 分配和回收速率不匹配	2
	非LEAK	27
无效地址或空指针解引用 (NPD)	N1. 指针使用前没有检查是否为空	10
	N2. 某些操作的执行取决于指针是否为 nil	1
	N3. 杂项	10
	非NPD	9
悬垂指针 (DP)	D1. CGO 和unsafe 机制	2
	D2.自行实现的垃圾回收机制	14
	非DP	34
合计		130

表 10 内存泄漏 LEAK 问题的人工分类

LEAK 问题类	LEAK 问题子类	Issues 数量	
L1. 资源未被显式释放	L1-1. 配对资源未被一起释放	2	8
	L1-2. 缺少对释放函数的调用	6	
L2. 不正确的显式释放	L2-1. 逻辑错误导致程序中的显式释放函数并没有被调用	2	5
	L2-2. 分配多次，释放一次	2	
	L2-3. 使用切片表达式前没有删除底层数组隐式引用	1	
L3. 并发	L3-1. 协程独占的数据结构过多	1	2
	L3-2. 并发阻塞导致协程资源泄漏	1	
L4. 依赖项内存泄漏	L4-1. 旧版本存在内存泄漏	4	5
	L4-2. 使用依赖项的默认参数时内存使用量较大	1	
L5. 拷贝大内存	L5-1. append 拷贝了内存占用量大的切片	1	1
L6. 分配和回收速率不匹配	L6-1. 不合理的回收机制导致待回收对象仍然被长期引用	1	2
	L6-2. 分配时复用较少	1	
总计		23	

代码 5 L1. 资源未被显式释放问题模式

```
1.  // L1-1 pingcap/tidb/issues/27353
2.   func (x *Resource) Close() {
3.  +  if x.pairResource != nil {
4.  +    x.pairResource.Close() // add close
    here
5.  +  }
6.     }
7.  // L1-2-defer grpc/grpc-go/issues/6642
8.    ticker := time.NewTicker(...)
9.  + defer ticker.Stop() // add Stop here
10. // L1-2-error pingcap/tidb/issues/34666
11.   if err != nil {
12. +  resource.Close() // add close here
13.    return err
14.   }
15. // L1-2-otherref btcsuite/btcd/pull/2105
16. + if c.batch { // fix here
17. +  c.batchList.Remove(element) // fix here
18. + } else {
19.    c.requestList.Remove(element)
20. + }
```

代码 6 L2. 不正确的显式释放问题模式

```
1.  // L2-1 grpc/grpc-go/issues /2444
2.  + resource.cancel()
3.    if someCondition {
4.      return // return too early
5.    }
6.  - resource.cancel() // resource not cancelled before
return
7.   // L2-2 kubernetes-sigs/cluster-api/issues/9542
8.  - resource = createResource()
9.   if someCondition {
10. -  resource = createResource() // create twice
11.  }
12. + resource = createResource()
13.   resource.stop() // stop once
14.  // L2-3 grpc/grpc-go/issues/4758
15. - item, q.queue = q.queue[0], q.queue[1:] // q.queue[0]
will not be used forever
16. + item = q.queue[0]
17. + q.queue[0] = nil
18. + q.queue = q.queue[1:]
19.    ... // destroy item after return
```

L2. 不正确的显式释放导致无效资源仍然被隐式引用 在该问题类中包含3个小类，模式代码如代码 6所示。

L2-1: 因逻辑错误导致资源的显式释放函数未被调用，引起内存泄漏。代码6中第1-5行展示了grpc-go的Issue#2444 [48]的代码简化，由于if条件的提前返回，导致资源的释放函数cancel未被调用，致使资源未被释放。

L2-2: 自行内存管理下分配次数和释放次数不匹配，导致内存泄漏。代码 6中第 7-13行展示了cluster-api的Issue#9542 [49]的代码简化，在某些执行路径下，资源会被创建两次，而最终资源释放时却只释放一次，这导致其中一份资源无法被正确回收，引起内存泄漏。

L2-3: 切片表达式[50]`s[start:end]`可能会使隐式引用未被删除，导致内存泄漏。Go使用标记-清扫方法回收不可达对象。当一个对象不再被任何指针所引用时，它可以被回收。Go语言引入切片类型，内部由包含数组指针、长度、容量的结构体来实现，其中数组指针指向底层数组中的某个元素。切片表达式会修改数组指针的位置，同时改变切片的长度和容量；但是它不会修改底层数组中对其他对象的引用关系。这导致使用切片表达式创建新切片之后，新切片相比原切片修改了数组指针的位置或者长度，无法通过索引表达式合法访问特定对象，但是底层数组对这些特定对象的引用仍然存在。这些特定对象即使之后不再被访问，也需要等到底层数组被垃圾回收器回收时才能被回收，其生命期被误延长了，导致内存泄漏。代码 6中第 14-19行展示了grpc-go的 Issue#4758 [7]的代码简化，原始代码尝试通过切片表达式`q.queue = q.queue[1:]`删除对`q.queue[0]`所指向对象的引用。然而实际上`q.queue[0]`仍然存在于新切片`q.queue`的底层数组中，造成隐式的内存引用，从而导致`q.queue[0]`引用的对象无法被及时回收，导致内存泄漏。

L3. 并发导致的内存占用量大 在该问题类中包含 2 个小类，模式如代码 7 所示。

L3-1: 协程独占的数据结构过多。在高并发场景中，当每个协程都独占一个数据结构时，每创建一个新协程就会为该数据结构分配内存。这种设计会导致内存占用不断增加，最终表现为内存泄漏。代码 7 中第 1-11 行展示了 tidb 的 Issue#13883 [51]的代码简化。在该问题中，sync.Mutex 和其他结构体域没有被设计为可以在多个协程之间共享，导致每个协程都分配自己的内存空间，造成内存使用量在高并发情况下急剧上升。

L3-2: 协程在阻塞时无法正常退出，也可能导致内存泄漏。代码 7 中第 12-16 行展示了 tidb 的 Is- sue#32412 [52]的代码简化。在该问题中，由于测试函数没有对创建的协程中的 channel 进行接收操作，导致协程阻塞在 channel 的发送操作上，无法正常退出，最终造成协程资源泄漏。

L4. 依赖项内存泄漏 依赖项的内存泄漏也是导致 Go应用程序内存泄漏的一个重要原因。在该问题类中包含2个小类，模式如代码8所示。

L4-1: 在使用外部依赖项时，如果依赖的库或版本存在内存泄漏问题，且未及时更新到修复版，可能会导致内存泄漏。问题通常出现在使用 Go 官方的依赖管理工具（go.mod）或 Git的子模块管理（git submodule）时，

没有注意到依赖项本身的内存问题。更新依赖库的版本，或切换到没有内存泄漏的版本，是解决此问题的有效途径。

L4-2: 在使用依赖项时，若没有调整默认的内存配置参数，也可能导致内存占用过高，从而引发内存泄漏。代码 8展示了pomerium的 Issue#4650 [53]的代码简化。问题修复通过配置 window size，来优化内存使用，减少不必要的内存占用。

代码 7 L3. 并发问题模式

```
1.  // L3-1 pingcap/tidb/issues/13883
2.    type Handle struct {
3.      ctx sessionctx.Context // can't be shared
4.  -   sync.Mutex // can be shared
5.      ... // can be shared
6.  +   *HandleShare
7.    }
8.    type HandleShare struct {
9.  +   sync.Mutex
10. +   ...
11.   }
12. // L3-2 pingcap/tidb/issues/32412
13.   go func() { ch <- 2 }
14.    ... // does not receive from ch before returning
15. + require.Equal(t, 2, <-ch)
16.   return
```

代码 8 L4. 依赖项内存泄漏问题模式

```
1.  // L4-2 pingcap/tidb/pull/12472
2.  - var zstdEncoder, _ = zstd.NewWriter(nil,
    zstd.WithEncoderLevel(zstd.SpeedBestCompression))
3.  + const zstdWindowSize = 8 << 10 // 8kiB
4.  + var zstdEncoder, _ = zstd.NewWriter(nil,
5.  +   zstd.WithEncoderLevel(zstd.SpeedDefault),
6.  +   zstd.WithWindowSize(zstdWindowSize),
7.  + )
```

代码 9 L5. 拷贝大内存问题模式

```
1.  // L5-1 pingcap/tidb/pull/12472
2.- historySlice = GetHistory()
3.- newSlice = append(newSlice, historySlice...) // copy large slice
4.+ historySliceIter = GetHistoryIter() // use iterator instead of slice
5.- for job := range newSlice {
6.+ for job := range historySliceIter.GetItem() {
7.+   ...
8.+ }
```

L5. append拷贝了内存占用量大的切片 在Go语言中，append操作可能会触发底层数组的重新分配特别是在处理包含大量元素的切片时，可能造成内存占用激增，最终表现为内存泄漏甚至导致内存耗尽 (Out of Memory, OOM)。代码 9展示了tidb的 Issue#12472 [54]的代码简化。在该问题中，append操作对historySlice这一包含大量元素的切片进行了拷贝，结果生成了两份内存占用量大的切片。由于系统需要同时维护这两份切片的数据，内存使用量迅速飙升，最终导致内存耗尽。

L6. 分配回收速率不匹配 在某些高性能 Go 应用（如数据库系统tidb[55]）中，开发者可能会自行实现分配和回收机制。然而由于分配和回收速率不匹配的问题，可能导致内存使用量持续增长，从而导致内存泄漏。该类问题模式如代码 10所示。

L6-1: 不合理的回收机制导致待回收对象仍然被长期引用。在自行实现的分配回收机制中，如果回收机制不合理，可能导致待回收对象长时间被引用，从而引发内存泄漏。但是如果缓冲池没有限制存储的对象数量，就会导致被缓冲池回收的对象总是存在引用，其生命期和缓冲池对象一致。而由于缓冲池的生命期通常贯穿整个程序生命期，造成了待回收对象在程序运行期间无法被及时回收，从而表现为内存泄漏。代码 10中第1-9行展示了tidb的 Issue#39331抽象后的问题及修复。问题发生在tidb自定义的kvMemBuf类型的缓冲池中，缓冲池通过Recycle方法回收对象，但是问题代码中未能正确处理缓冲池中的对象回收。具体来说，availableBufs对buf进行了持久引用，导致每

代码 10 L6. 分配和回收速率不匹配问题模式

```
1.  // L6-1 pingcap/tidb/issues/39331
2.   func (mb *kvMemBuf) Recycle(buf *bytesBuf) {
3.     ...
4.  +  if len(mb.availableBufs) >= maxAvailableBufSize {
5.  +    mb.availableBufs = mb.availableBufs[1:]
6.  +  }
7.     ...
8.     mb.availableBufs = append(mb.availableBufs, buf)
9.   }
10. // L6-2 pingcap/tidb/issues/39331
11.  func (mb *kvMemBuf) AllocateBuf(size int) {
12. +   existingBuf := findAvailableBuf(mb.availableBufs)
13. -   if strictConstraint {
14. -     mb.buf = mb.availableBufs[0]
15. +   if existingBuf != nil {
16. +     mb.buf = existingBuf
17.     } else {
18.       mb.buf = newBytesBuf(size)
19.     }
20.  }
```

次调用时，buf所引用的对象无法及时被垃圾回收器回收，直到availableBufs也不再被引用为止。由于缓冲池生命期较长，回收过程较慢，最终导致了内存泄漏。

L6-2: 分配时复用较少。在某些内存分配管理策略中，当分配新内存时，如果没有充分利用已有的资源，会导致大量的内存浪费。具体来说，如果在分配缓冲区时，仅当第一个可用缓冲区满足特定条件时才会进行复用，而忽略了其他可用缓冲区的情况，结果就会导致新分配的缓冲区数量增加，进一步导致内存占用量过大。代码 10 中第 10-20行中，修复前的代码采用了如上所述的策略，造成了较大的空间浪费。而修复后的代码会在所有可用缓冲区进行检查，只有找不到合适的缓冲时才会新分配，大幅提高内存复用效率，避免不必要的浪费。

结论 4 - RQ2

内存泄漏问题在 Go 语言中仍然广泛存在，并且种类繁多。虽然 Go 语言实现了自动垃圾回收机制，资源的释放仍然需要开发者手动添加，同时 defer 延迟机制的引入并没有从根本上解决内存泄漏的问题。在内存和其他资源释放函数的调用中，依然存在逻辑错误、调用次数不准确以及误用的情况。特别是在高并发环境下，因并发阻塞及不同协程间共享数据结构设计不当，可能导致内存占用异常增大。此外，开发者在自行实现内存分配和回收机制时，由于垃圾回收的存在，开发者更容易在自行实现的分配回收机制中无意间引入对可回收对象的长生命期引用，导致这些对象在运行时无法被垃圾回收机制及时清理。

4.3.2 无效内存地址或空指针解引用问题模式

本文采样得到了30个Issues，其中有21个确定为无效内存地址或空指针解引用问题。本节对这 21个Issues进一步分析。结果显示，在Go语言中，无效内存地址或空指针解引用问题主要表现为如表11所示的类别及数量分布。

表 11 无效内存地址或空指针解引用 NPD 问题的人工分类

NPD 问题类	NPD 问题子类	Issues 数量	
N1. 指针使用前没有检查是否为空	N1-1. 函数返回值指针在使用前没有检查是否为空	6	10
	N1-2. 条件表达式使用指针前没有判断指针是否为空	2	
	N1-3. 指针变量在声明后没有初始化就直接使用	1	
	N1-4. 使用指针前非空判断条件不充分	1	
N2. 某些操作的执行取决于指针是否为 nil	N2-1. 当错误发生后指针没有被置为 nil 以无效化后续操作	1	1
N3. 杂项	N3-1. 杂项	10	10
总计		21	

N1. 指针使用前没有检查是否为空 在该类别中，根据该问题发生的上下文可分为如下四小类，模式如代码 11所示。

N1-1: 未对函数调用返回的指针进行空值检查。代码 11中第1-4行展示了cockroach 的 Issue#48092 [56]的代码简化，其中asString是函数调用返回的指针。当asString为空时，编译器生成的运行时检测代码会发现空指针并导致提前崩溃。虽然这种运行时检查避免了程序继续执行，防止了未定义行为，但它实际上可以通过静态分析工具或者编译器的编译时检查进行提前检测，提前发现部分空指针解引用问题。尽管如此，表 11中仍有 6 条因未检查函数返回指针是否为空而导致崩溃的 Issues。

N1-2: 在条件表达式中使用指针前未进行空值检查。代码 11中第 5-8行展示了milvus的Issue#17823 [57]的代码简化。其中，ptr在未检查是否为空的情况下就直接进行了方法调用，导致崩溃。

N1-3: 指针变量在声明后未初始化就直接使用。代码 11中第9-15行展示了milvus的 Issue#5333 [58]的代码简化，由于返回值变量ptr未初始化就进行解引用访问操作ptr.ID，导致程序运行时崩溃。

N1-4: 使用指针前非空判断条件不充分。代码11中第16-20行展示了milvus的Issue#31495[59]的代码简化。由于len(...) == 0的条件判断不等价于obj2不为空，导致在进行解引用访问obj2.field时发生崩溃。

N2. 某些操作的执行取决于指针是否为 nil 指针是否为nil可以作为是否执行与指针相关操作的条件。在某些情况下，指针应该被置为nil以防止非法操作被调用。代码13展示了修复代码，当发生错误时，d.serializer被置为空，以防止后续对非法操作的调用。

N3. 杂项 除了上述提到的简单模式之外，在21条与无效内存地址或空指针解引用相关的 Issues中，有10条属于该范畴。由于这些问题的修复规模较大并且需要较强的领域知识，故无法推断出导致问题的核心原因。

代码 11 N1. 指针使用前没有检查是否为空问题模式

```
1. //   N1-1 cockroachdb/cockroach/issues/48092
2.    asString = GetAString() // may return nil
3. + if asString == nil { return }
4.    parse(*asString) // asString not checked for nil
5. // N1-2 milvus-io/milvus/issues/17823
6. // ptr not checked for nil
7. - return ptr.GetState() != someState && ...
8. + return ptr != nil && ptr.GetState() != someState
      && ...
9. // N1-3 milvus-io/milvus/issues/5333
10.  func getID() (ptr *Meta, err error) {
11.      // ptr is not initialized
12. +    ... // initialize ptr
13.      ptr.ID = ...
14.      ...
15.  }
16. // N1-4 milvus-io/milvus/issues/31495
17.  if len(GetObj2ByObj1(obj1)) == 0
     { continue }
18.  obj2 = GetObj2byObj1AndObj3(obj1, obj3)
19. + if obj2 == nil { continue }
20.  obj2.field // obj2 can still be nil
```

代码 12 N2. 某些操作的执行取决于指针是否为 nil 问题模式

```
1.  // N2-1 cockroachdb/cockroach/pull/59477
2. + defer func() {
3. +   if err != nil {
4. +     // If an error occurs, set the serializer to nil
      to avoid any future attempts to call other
      methods.
5. +     d.serializer = nil // fixed here
6. +   }
7. + }()
8.   if err := d.serializer.Finish(); err != nil {
9.     return err
10.  }
```

结论 5 - RQ2

空指针解引用问题在 Go 语言中仍然存在。尽管 Go 语言在运行时添加了空指针解引用的检查操作以确保程序安全，但是在开源社区中仍然存在大量因指针未检查空值而导致程序崩溃的 Issues。针对这类问题，建议在代码审查中使用静态分析工具，如 nilaway[60]和 nilness[61]，检查是否存在空指针解引用的问题。

4.3.3 悬垂指针问题模式

本文人工分析了50条通过搜索“dangling pointer”关键词和采样得到的Issues。在这些Issues中，只有2条Issues[62,63]和悬垂指针直接相关，并且它们是由于CGO(Go语言调用C语言的机制)和unsafe机制导致的。其余的Issues仅在内容和评论中提及了dangling关键词。大多数情况下，包含这些Issues的应用引入了自定义的引用系统，如kubernetes实现了集群资源的引用系统，并实现了更高层次的垃圾回收[64]，用于清理集群资源；版本控制工具git-lfs和 pachyderm实现了文件对象的引用系统，通过在文件系统中存储objects来表示版本信息。在这些引用系统中，悬垂对象通常是引用系统中更高抽象层次的对象，例如集群资源对象、commit对象等，需要程序员手动维护引用关系的有效性。

结论 6 - RQ2

悬垂指针问题在 Go 语言中并不常见，Go 语言较好地解决了悬垂指针问题。

4.4 内存安全问题的修复

为了了解内存安全问题的修复难度与修复成本，本文进一步统计了包含有效pull request的Issues的修复时间、修复规模。统计方法详见第2.2节。

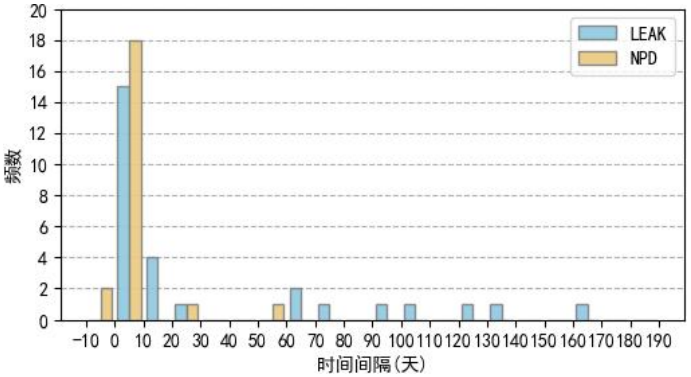

图 5 修复时间间隔直方图

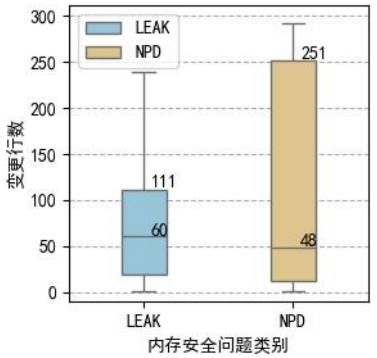

图6 代码变更行数箱型图

修复时间间隔 本文对所有人工分析的内存安全问题模式进行了修复时间的统计，其修复时间间隔的统计直方图如图 5所示。

在内存泄漏问题中，大部分问题在30天之内被修复，但也有8个问题需要至少60天才能解决。相比之下，无效地址或空指针解引用问题大多在10天之内修复，只有一个问题需要50-60天的时间才能解决。整体而言，无效地址或空指针解引用问题的修复时间较短。另外，有两个Issues[65,66]的修复时间为负数，即2个PR在Issue被创建的5天前已提交修复。经过人工审查发现，有评论提到在出现时间更早的PR修复后的代码版本中，Issue所提及的问题已无法复现，可能已被修复。因此，无效地址或空指针解引用问题通常能够较快得到解决。

修复代码规模 本文对所有人工分析的内存安全Issues/PRs进行了修复代码规模的统计。修复代码变更行数箱型图如图6所示。其中内存泄漏问题的修复代码变更中位数为 60行，无效地址或空指针解引用问题的修复代码变更中位数为48行。但是从75%分位数来看，内存泄漏问题的修复代码变更规模为111行，相比空指针解引用问题的修复规模251行要小。

结论 7 - RQ2

内存泄漏问题通常在 30 天内被修复，但是也有个别问题修复时间超过 60 天。无效地址或空指针解引用问题通常在 10 天内修复，修复速度较快，甚至出现因版本未及时更新而导致重复提出 Issue 的情况。从修复规模来说，两类问题的修复变更代码行数中位数均为数十行，修复规模相对较小。

5 RQ3: 内存问题的自动检测

针对第 4 节总结的一系列内存问题模式，如何自动检测它们仍然面临挑战。本节给出一些可能的自动检测方案尝试解决这个问题。

5.1 一些问题模式的自动检测方法可能性

要实现自动检测，问题模式必须足够简单，能够通过基础的过程内数据流分析算法进行检测。然而对于跨多个函数的问题，即使能够实现自动检测，也会由于分析时间过长或者误报过多导致实用性较低。此外，当开发者在应用中自定义了内存分配与回收算法时，检测方法需要专门为该应用设计，难以广泛适用于其他Go项目。

对于L1模式来说，自动检测首先需要识别资源类型。这需要通过人工标注或者其他自动的方式实现资源类型的标注。假定定义了`Open`、`Close`方法的结构体类型为资源类型。对于L1-1模式，可以检测资源类型定义中是否引用了其他资源类型，如果引用，进一步检查`Close`方法中是否调用了其他资源的`Close`方法判断代码是否存在问题。阻碍该问题自动化检测的难点在于不同资源类型的申请和释放函数可能有不同的函数名称和签名，需要通过标注识别，并且需要识别不同结构体之间的引用关系。对于L1-2模式，可以通过使用描述资源打开或者关闭状态的状态机以及前向分析判断被打开的资源是否在程序结束时一定被关闭。

对于L2-1, L2-2模式，这些模式涉及较多的分支语句，可能需要引入路径敏感性，为资源对象添加条件属性，匹配特定条件下资源释放的执行情况。对于L2-3模式，本文使用*QLStat*进行了检测，将在后文中详细描述该方法。

对于L3-1模式，需要分析在不同协程中，是否存在特定的数据结构是可以被共享的。阻碍该模式自动化检测的难点在于首先需要识别在不同协程之间共享的类型，然后需要识别类型中哪些部分是协程独占的，哪些部分是可以被多协程共享的。对于阻塞导致的协程泄漏L3-2模式，可以采用并发阻塞相关的检测工具进行检测，例如GCatch[8]等工具。

对于L5-1模式，需要匹配append函数调用的最后一个实参是否为slice...模式（其中...用于将slice赋值给可变参函数的形参），评估slice中所关联的内存大小。如果slice对象所关联的内存较大，则需要给出警告。阻碍该模式自动化检测的难点在于如何给出slice对象所关联内存大小的估计。

对于N1-1模式，需要检查指针类型的返回值接收者是否在使用前进行了非空检查。对于N1-2模式，需要检查条件表达式中在指针变量的方法调用之前是否进行了非空检查。对于N1-3模式，需要检查指针类型的函数返回值是否在使用前进行了初始化。这三类模式能较好进行检测的难点在于总是存在情况静态分析无法得知指针是否为空，从而产生误报。

对于未提及的模式，要实现自动检测需要将函数语义编码到自动检测工具中，同时还需要考虑控制流的复杂性，因而自动检测的难度较大。

接下来，将介绍本文如何处理L2-3问题模式。

5.2 内存问题自动检测工具实践

根据Staticcheck的规则说明, go vet的帮助文档, golangci-lint的linter文档以及其他代码检查工具的功能描述，目前并没有现成的规则用于检测L2-3模式。由于该模式形式简单，修复的代价较小，同时可能造成内存泄漏，因此本文认为有必要对其进行检测。本文基于*QLStat*实现了对L2-3问题模式的自动检测工具，并在评测中对966个项目数据库进行了检查，最终报告了574处问题。在574处问题中，人工审查了70个问题，最终发现6个可能正确的报告，并通过GitHub Issue进行了反馈，最终有1个问题被开发者确认为可能存在内存泄漏的问题。

5.2.1 设计与实现

图 7 展示了 L2-3 问题模式更加一般化的形式，其中左半部分为正确的代码模式，代码检查工具不应该报告；右半部分为可能存在内存泄漏问题的代码模式。L2-3 问题模式的本质原因在于：切片表达式 s2[start:]中的被切部分 s2[0:start]所引用的对象（即被切对象）由于始终被 s2 的底层数组所引用，导致无法被及时回收。如果在 s3 = s2[start:]赋值之前，可以将 s2[0:start]及时设置为 nil，就可以避免被切对象的回收需要延迟到 s2 底层数组被回收的时刻。

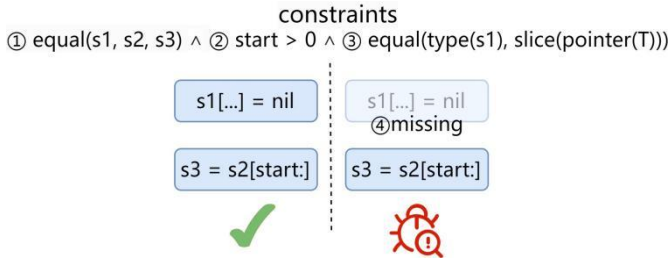

图 7 L2-3 代码检查工具规则

代码 13 L2-3 应消除的误报示例

```
1. x := a.f
2. x[0] = nil
3. a.f = a.f[start:]
```

根据上述原因，为了检测图 7右半部分的模式，本文设计了相应的代码检查工具。其使用的约束如图 7中数字标号标识的约束所示：(1) s1~s3三个slice值应该相等，即使用相同的slice，(2) 切片开始位置start应该大于 0，(3) s1切片的底层元素类型应该为指针类型，(4) s1[...] = nil和s3 = s2[start:]不应该同时出现。在具体实现中，针对约束1，采用了全局值编码 GVN[67] 以及SSA[68] 的特性，以尽可能多地识别运行时值相同的表达式。针对约束4，采用“是否出现在相同基本块中”作为分析粒度。

利用以上约束，代码检查工具在检测代码 13时，不会报告这段代码有问题。这是因为在同一基本块中存在对元素x[0]设置nil的操作，这是正确的做法。另外，代码检查工具也识别了x和a.f在运行时是相同的值，减少误报现象的发生。

代码 14 aws-sdk-go 可能存在的内存泄漏问题

```
1. // L2-3 aws/aws-sdk-go/issues/5327
2. // Next will use the S3API client to iterate through a
     list of objects.
3. func (iter *DeleteListIterator) Next() bool {
4.   if len(iter.objects) > 0 {
5.     iter.objects = iter.objects[1:]
6.   }
7.   ...
8.   return len(iter.objects) > 0
9. }
```

5.2.2 效果评估

在实现上述工具后，本文使用该代码检查工具对966个项目数据库进行了检查，并最终报告了574处问题，平均每个仓库报告了0.59个问题。

为了进一步评估工具的准确率，在这些报告中，作者人工审查了70个报告。发现其中有6个为可能正确的报告，有64个为错误的报告。在人工审查时，作者制定了如下评估标准：

1.如果有其他对象引用被切对象，那么：

(a) 如果这个对象逃逸出函数作用域，则报告错误。

(b) 否则按照没有对象引用被切对象的情况处理。

2.如果没有其他对象引用被切对象，那么：

(a) 如果切片表达式之后的执行路径可能到达append函数，则报告错误。

(b) 如果没有append函数，则报告正确。

上述标准中，不同的报告有不同的原因：1(a) 如果引用被切对象的对象o逃逸，那么被切对象的生命期实际上由o决定，设置nil操作意义不明确，报告错误。2(a) 如果切片表达式后面的执行路径可能到达append函数，那么slice底层数组会扩容，slice底层数组在进行扩容时会丢弃对原始数组的引用，因此被切对象将被及时回收，报告存在错误。这里只需要考虑切片的append操作。这是因为Go语言规范[33] 中定义的其他开发者可见的slice操作（包括索引表达式、切片表达式、copy函数、clear函数）都只能在slice所定义的区间内进行修改、读取操作，不会让slice引用一个新底层数组并丢弃对旧底层数组的引用，也就不会丢弃对被切对象引用，从而可能引发内存泄漏，需要被报告出来。2(b) 如果切片表达式之后的执行路径不可能到达append函数，那么slice底层数组不会扩容，被切对象仍然会被slice底层数组引用，因此被切对象无法被及时回收，报告正确。

虽然这70个报告中有较多的误报，但是仍然有6个报告是正确的。这表明代码检查工具具有实际效用，同时也表明L2-3模式在真实世界项目中依然存在，且有进一步优化的空间。此外，人工分析的标准实际上可以作为改进代码检查工具的参考，通过引入指针分析、逃逸分析以及过程间控制流分析等分析技术加强对该问题的检查，从而及时报告这类导致内存泄漏的代码模式。但引入这些技术也会带来较高的代价。这里的分析代价主要有几方面因素构成：1) 需要借助于逃逸分析，这对于CodeQL来说是困难的。2)需要通过静态分析判断执行路径是否可append函数，而这可能导致不准确。此外，这里对append函数模式也有要求，即只有在形如s = append(s, _) 的模式下，s会丢弃对扩容前底层数组的引用，此时才报告正确。而 s = append(s, _)的两个s可能是别名关系，而别名关系判断比较困难，可能存在漏报的情况，即部分本来是别名的对象没有被识别为别名，进而导致s = append(s, _)模式没有被识别出来，最终导致误报。

此外，针对6个人工识别为正确的报告，本文在GitHub上提交了5个Issues，并已收到1个肯定的回复[69]。问题如代码 14所示，发生在删除列表迭代器往后遍历的过程中，没有清除对`iter.objects[0]`所指向对象的引用，从而可能导致内存泄漏。在该回复中，仓库贡献者确认了设置nil并不会造成错误的影响，在高负载情况下可能导致内存泄漏。这进一步表明，L2-3问题模式在真实世界Go项目中仍然存在。

6 有效性讨论

内部有效性威胁 在使用*QLStat*统计不同模式的分布时，由于语法树结点的多样性，可能出现漏扫漏查或多扫多查的情况。为缓解该问题带来的影响，本文采用了CodeQL这一相对更加可靠的工具进行统计分析，同时在编写查询语句时，作者遍历了CodeQL中和Go语言语法树结点相关的文档，并添加详尽的测试以确保查询语句的准确性。另外由于Go语言的语法糖，某些访存操作可能会被遗漏。针对这些问题，作者仔细查阅了Go语言的规

范[33]，并在统计相关操作时包含了语法糖所隐藏的操作。例如在统计解引用操作Deref(*a)时，除了统计显式解引用操作之外，还统计了语法糖`a.f`中包含的隐式解引用操作。

在统计连续域访问和连续解引用操作时，由于查询语句底层基于抽象语法树实现，*QLStat* 在对形如`a.f.g`和`**p`这类具有语法结构递归嵌套的模式进行查询时，未考虑别名关系。因此，形如`x = a.f; y = x.g`以及`x = *p; y = *x`等通过别名引入的连续域访问和连续解引用操作，会被视为多个独立的单次域访问或者解引用操作。为了缓解这个问题，需要利用CodeQL解析器生成的SSA表示[68] 识别变量的定义-使用（def-use）关系，从而捕获上述两个例子中出现的别名，进而识别含别名的连续域访问和连续解引用操作。此外，对于更复杂的别名情况，例如x = &a.f, y = (*x).g，由于 CodeQL缺少Go语言的别名分析支持，因此当前建立在CodeQL之上的*QLStat*仍无法统计这类包含更复杂情况的连续操作模式。这使得统计得到的连续操作数量相比于真实数量偏少。

由于 GitHub REST API的限制，每次Issue搜索最多只能获取1000个 Issues。这就导致本文无法获取到所有的Issues，可能会影响统计结果。然而，统计数据显示，达到1000个 Issues的仓库只有17个仓库，因此基于此的统计分析结果仍然是可靠的。在分析总结问题模式时，由于模式是人工总结并分类的，因此存在分类不准确的情况。此外，使用代码检查工具检测L2-3模式时，也存在一定的局限性和有效性威胁。静态分析的结果无法完全反映程序的实际运行状态，且可能存在误报和漏报的情况。由于slice的别名问题、数据流、控制流的复杂性，针对L2-3 问题模式的代码检查工具难以实现完全准确的报告，可能存在漏报误报的情况。对此，本工作在检测中已使用如GVN, SSA等方式分析别名情况，以减少漏报，同时通过检测在同一基本块中不存在对元素设置为nil操作的约束减少误报，最终将报告数量控制在每个仓库最多23个报告，中位数2个报告的范围内。

在统计PR修复时间间隔时，本文将PR的关闭时间作为修复结束时间。这可能无法真实反映合并时间。如果一个PR被合并，则该PR的关闭时间、被合并的时间以及修复结束时间是一致的。但是如果一PR没有被合并就关闭了，那么此时应假设Issue的关闭时间为修复结束时间。因此“使用PR的关闭时间作为修复结束时间”引入的有效性威胁在于可能会导致对应PR没有被合并的Issue的修复时间被低估。但是在大多数PR被合并的条件下，以PR关闭时间作为修复结束时间的统计仍具有参考价值。

外部有效性威胁 本文分析的仓库并不能完全代表现实世界Go项目中的所有内存性能与安全问题。然而本文选择过去一年内更新的 996个项目以反映现实世界中仍然活跃的项目中的内存性能与安全问题。另外，由于通过关键词在GitHub中搜索Issues/PRs只会搜索包含关键词的Issues/PRs，因此这些Issues/PRs并不能完全反映内存安全问题的全貌，在这些Issues/PRs上的分析也并不能完全覆盖对所有潜在问题。

在人工分析三类关键词的Issue时，NPD问题只分析了30条。这可能导致NPD内存问题模式不足以反映真实世界的情况。NPD问题的数量相较于其他问题Issue更少的原因在于NPD问题的发生主要和应用程序的特征有关。部分应用程序的NPD问题的上下文较为复杂。由于缺乏领域知识，较难总结NPD问题的模式。

7 相关工作

内存安全问题检测与修复 根据谷歌发布的报告[70]，内存相关问题有50多年的历史，通常会导致系统运行崩溃或者带来安全风险。例如内存泄漏、空指针解引用问题可能会被攻击者利用，通过程序崩溃来进行拒绝服务攻击[71]。至今还有很多研究在试图缓解内存问题带来的影响。很多研究已经在C, C++, Python, Rust等编程语言的内存安全问题方面开展，例如C/C++程序的内存泄漏检测[72–74]，C/C++程序的并发安全[75]，Python程序中的内存泄漏问题[76]，Rust在语言层次设计的所有权[77] 和借用机制以防止double free等问题，Rust语言层次设计的生命期标注[78]和Drop机制以及时释放资源，Rust程序上的内存问题和并发安全实证研究[79]等。相比之下，Go语言的显著特点是原生支持并发，这使得许多Go语言相关的研究聚焦于并发安全问题。相关工作包括对真实世界中Go语言项目存在的并发问题的实证分析[80]，通过程序验证[81,82]、静态分析[83]、模糊测试[84] 方式检测和修复Go语言程序的数据竞争、死锁问题。另外由于Go语言是类型安全的语言，而unsafe package会破坏类型安全机制，已有实证研究探索了unsafe package在真实世界中使用情况以及潜在危害[41,85]。但是对除并发安全、类型安全之外，针对Go语言中其他内存问题的研究工作相对较少。例如对于内存泄漏问题来说，在Go语言相关的研究工作中只有Saioc等[86] 通过检测部分死锁的go协程goroutine来检测内存泄漏问题。CodeQL分析的已有工作[87,88]主要聚焦于漏洞检测。本工作的创新之处在于利用CodeQL这种将程序代码解析存储到数据库中，然后使用查询语言进行查询得到代码查询结果的范式，实现对现实世界Go项目中内存相关基本操作分布情况的统计分析。同时基于这一方法，本工作对大量真实世界Go项目进行了L2-3模式的检查，给出可能存在内存泄漏问题的报告。

基于静态代码模式的优化 特定的静态代码模式可能造成性能问题。对这些模式（包括静态分析所建模的语句）的

统计分析有助于指导静态分析和程序优化的方向，从而缓解性能问题。例如 He等[89]基于三类普遍存在的容器使用模式，识别可能影响分析精度的依赖上下文的对象，在分析时只为这些对象添加上下文，而不为其他对象添加，从而在接近相同的精度下显著提升指针分析速度。本文对连续域访问次数和连续解引用的次数也表明：在分析这两类模式时，仅对单次操作进行比较准确的建模即可获得较大的收益，对于多次操作可以采取保守的建模方式以获得较快的分析速度。Selakovic等[10]的工作对JavaScript中导致性能问题的常见静态代码模式进行了总结，指导应用开发者避免常见的错误，指导研究者开发相应的性能工具，指导JavaScript引擎开发者处理这些流行的性能瓶颈模式。本文的第3节针对Go语言中的基本访存操作进行了建模，发现域访问操作和解引用操作在所有基本访存操作中合计占约42%，是潜在的性能优化点，需要在程序建模时重点关注。进一步分析表明，在所有转换为接口的模式中，赋值语句和调用语句中包含接口转换的模式最多，约占98%。同时，在这些转换接口的模式中，被转换类型为非指针类型的模式占34.96%。这些模式可能存在隐式堆内存分配，内存分配优化可以重点关注这些模式。

8 总结

本文提出了一套针对GitHub中开源Go仓库的实证分析框架，并利用该框架对GitHub中筛选出来的996个Go语言仓库中的内存访问模式、内存安全问题以及修复模式进行了实证分析与总结。

首先，针对Go语言的内存访问模式，本文使用*QLStat*对仓库进行了自动统计与分析，分析结果显示解引用和域访问操作分别占17.63%和25.44%的比例。这一结果说明，在进行编译器优化和静态分析工具建模时，必须重点关注这两类操作。进一步分析发现，大多数的域访问和解引用操作均为单次访问。与此同时，针对Go独有的interface类型，分析表明，在转换为接口的模式中，assign模式和call模式占据了绝大多数，在分析接口相关的内存分配时需要着重考虑。

其次，本文通过人工分析130个Issues/PRs，并留下44条确定为内存问题的Issues/PRs。这些Issues的类别分为内存泄漏、无效地址或空指针访问、悬垂指针三类。进一步的模式分析显示，内存安全问题在开源Go语言项目中频繁出现，并且最常见的类型是无效地址或空指针访问问题，而开发工具、云/微服务、容器/虚拟机等领域的项目更容易发生内存安全问题。此外，针对每一类的内存安全问题，本文也进行了模式的细分。

最后，针对模式细分中的L2-3问题模式(即由切片表达式赋值s3 = s2[start:]导致内存泄漏的问题模式)，本文使用*QLStat*框架开发了代码检查工具，分析了966个项目数据库，报告了574处问题。在这些问题中，人工审查了70个问题，最终发现6个可能正确的报告，并通过GitHub Issues提交给相关仓库，最终有1个被开发者确认为可能存在内存泄漏。

References:

[1] Go. Why Go > Case Studies . 2024. https://go.dev/solutions/case-studies. Accessed: 2024-12-1.

[2] Go. proposal: arena: new package providing memory arenas. 2022. https://github.com/golang/go/issues/51317. Accessed: 2024-12-1.

[3] Wang C, Zhang M, Jiang Y, et al. Escape from Escape Analysis of Golang. 42nd ACM/IEEE International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). 2020: 142-151.

[4] Ding B, Li Q, Zhang Y, et al. MEA2: a Lightweight Field-Sensitive Escape Analysis with Points-to Calculation for Golang . OOPSLA 2024: The Object-Oriented Programming, Systems, Languages, and Applications. Pasadena, California, USA, 2024.

[5] Peng H, Zhang Y, Ernst M D, et al. GoFree: Reducing garbage collection via compiler-inserted freeing . CGO 2025: International Symposium on Code Generation and Optimization. Las Vegas, NV, USA, 2025.

[6] grpc. grpc-go. 2024. https://github.com/grpc/grpc-go. Accessed: 2024-12-1.

[7] grpc. grpc/grpc-go/issues/4758. 2024. https://github.com/grpc/grpc-go/issues/4758. Accessed: 2024-12-1.

[8] Liu Z, Zhu S, Qin B, et al. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems . 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York, NY, USA: Association for Computing Machinery, 2021: 616–629. https://doi.org/10.1145/3445814.3446756.

[9] Veileborg O H, Saioc G V, Møller A. Detecting Blocking Errors in Go Programs Using Localized Abstract Interpretation . Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. DOI: 10.1145/3551349.3561154.

[10] Selakovic M, Pradel M. Performance Issues and Optimizations in JavaScript: An Empirical Study. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). 2016: 61-72. DOI: 10.1145/2884781.2884829.

[11] Chabbi M, Ramanathan M K. A Study of Real-World Data Races in Golang. 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI). 2022: 474-489.
[12] D.Honnef. Staticcheck - The advanced Go linter . 2024. https://staticcheck.dev/. Accessed: 2024-12-1.
[13] Go. go vet package. 2024. https://pkg.go.dev/cmd/vet. Accessed: 2024-12-1.
[14] GolangCI. golangci-lint. 2024. https://golangci-lint.run/. Accessed: 2024-12-1.
[15] Secure Go. gosec. 2024. https://github.com/securego/gosec. Accessed: 2024-12-1.
[16] Gechev M. revive. 2024. https://github.com/mgechev/revive. Accessed: 2024-12-1.
[17] Park Y G, Goldberg B. Reference escape analysis: optimizing reference counting based on the lifetime of references. SIGPLAN Notices, 1991, 26(9):178–189. https://doi.org/10.1145/115866.115883.
[18] Park Y G, Goldberg B. Escape Analysis on Lists. PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 1992: 116–127. https://doi.org/10.1145/143095.143125.
[19] Go. A Guide to the Go Garbage Collector. 2023. https://go.dev/doc/gc-guide.
[20] Jump M, McKinley K S. Cork: dynamic memory leak detection for garbage-collected languages. Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2007: 31-38.
[21] Go. The Go Programming Language Specification - Interface types . 2024. https://go.dev/ref/spec#Interface_types. Accessed: 2024-12-1.
[22] Go. The Go Programming Language Specification - Slice types . 2024. https://go.dev/ref/spec#Slice_types. Accessed: 2024-12-1.
[23] Go. The Go Programming Language Specification - Map types . 2024. https://go.dev/ref/spec#Map_types. Accessed: 2024-12-1.
[24] CockroachDB. cockroach. 2024. https://github.com/cockroachdb/cockroach. Accessed: 2024-12-1.
[25] Marr H. Making a Go program run 1.7x faster with a one character change . 2022. https://hmarr.com/blog/go-allocation-hunting/. Accessed: 2024-12-1.
[26] Honnef D. SA4031 - Checking never-nil value against nil . 2024. https://staticcheck.dev/docs/checks/#SA4031. Accessed: 2024-12-1.
[27] Honnef D. S1009 - Omit redundant nil check on slices, maps, and channels . 2024. https://staticcheck.dev/docs/checks/#S1009. Accessed: 2024-12-1.
[28] Honnef D. S1020 - Omit redundant nil check in type assertion . 2024. https://staticcheck.dev/docs/checks/#S1020. Accessed: 2024-12-1.
[29] Honnef D. S1031 - Omit redundant nil check around loop . 2024. https://staticcheck.dev/docs/checks/#S1031. Accessed: 2024-12-1.
[30] Honnef D. SA5011 - Possible nil pointer dereference . 2024. https://staticcheck.dev/docs/checks/#SA5011. Accessed: 2024-12-1.
[31] GitHub. Codeql. 2024. https://codeql.github.com/. Accessed: 2024-12-1.
[32] GitHub. Ql language reference. 2024. https://codeql.github.com/docs/ql-language-reference/.
[33] Go. The Go Programming Language Specification. 2024. https://go.dev/ref/spec. Accessed: 2024-12-1.
[34] The MITRE Corporation. CVE . 2024. https://cve.mitre.org/. Accessed: 2024-12-1.
[35] The MITRE Corporation. Common Weakness Enumeration: CWE . 2024. https://cwe.mitre.org/index.html. Accessed: 2024-12-1.
[36] Go Security team. Go Vulnerability Database . 2024. https://pkg.go.dev/vuln/. Accessed: 2024-12-1.
[37] The MITRE Corporation. CWE - Helpful Views . 2024. https://cwe.mitre.org/data/index.html#helpful_views. Accessed: 2024-12-1.
[38] The MITRE Corporation. CWE VIEW: Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses . 2024. https://cwe.mitre.org/data/d efinitions/1387.html. Accessed: 2024-12-1.
[39] The MITRE Corporation. CWE VIEW: Weaknesses in Software Written in C++ . 2024. https://cwe.mitre.org/data/definitions/659.html. Accessed: 2024-12-1.
[40] Go. go/src/runtime/panic.go. 2024. https://github.com/golang/go/blob/master/src/runtime/panic.go. Accessed: 2024-12-1.
[41] Costa D E, Mujahid S, Abdalkareem R, et al. Breaking Type Safety in Go: An Empirical Study on the Usage of the unsafe Package. IEEE Transactions on Software Engineering, 2021, 48(7):2277-2294. DOI: 10.1109/TSE.2021.3057720.
[42] GitHub. REST API endpoints for search. 2022. https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28. Accessed: 2024-12-1.
[43] PingCAP. pingcap/tidb/issues/27353. 2024. https://github.com/pingcap/tidb/issues/27353. Accessed: 2024-12-1.
[44] grpc. grpc/grpc-go/issues/6642. 2024. https://github.com/grpc/grpc-go/issues/6642. Accessed: 2024-12-1.
[45] grpc. Ticker leak in weighted round robin picker creation. 2023. https://github.com/grpc/grpc-go/issues/6642#issuecomment-1723821389.

Accessed: 2024-12-1.

[46] PingCAP. pingcap/tidb/issues/34666. 2024. https://github.com/pingcap/tidb/issues/34666. Accessed: 2024-12-1.

[47] btcsuite. btcsuite/btcd/pull/2105. 2024. https://github.com/btcsuite/btcd/pull/2105. Accessed: 2024-12-1.

[48] grpc. grpc/grpc-go/issues/2444. 2024. https://github.com/grpc/grpc-go/issues/2444. Accessed: 2024-12-1.

[49] Kubernetes SIGs. kubernetes-sigs/cluster-api/issues/9542. 2024. https://github.com/kubernetes-sigs/cluster-api/issues/9542. Accessed: 2024-12-1.

[50] Go. The Go Programming Language Specification - Slice expressions . 2024. https://go.dev/ref/spec#Slice_expressions. Accessed: 2024-12-1.

[51] PingCAP. pingcap/tidb/issues/13883. 2024. https://github.com/pingcap/tidb/issues/13883. Accessed: 2024-12-1.

[52] PingCAP. pingcap/tidb/issues/32412. 2024. https://github.com/pingcap/tidb/issues/32412. Accessed: 2024-12-1.

[53] Pomerium. pomerium/pomerium/pull/4650. 2024. https://github.com/pomerium/pomerium/pull/4650/files. Accessed: 2024-12-1.

[54] PingCAP. pingcap/tidb/pull/12472. 2024. https://github.com/pingcap/tidb/pull/12472. Accessed: 2024-12-1.

[55] PingCAP. Tidb. 2024. https://github.com/pingcap/tidb. Accessed: 2024-12-1.

[56] CockroachDB. cockroachdb/cockroach/issues/48092. 2024. https://github.com/cockroachdb/cockroach/issues/48092. Accessed: 2024-12-1.

[57] Milvus. milvus-io/milvus/issues/17823. 2024. https://github.com/milvus-io/milvus/issues/17823. Accessed: 2024-12-1.

[58] Milvus. milvus-io/milvus/issues/5333. 2024. https://github.com/milvus-io/milvus/issues/5333. Accessed: 2024-12-1.

[59] Milvus. milvus-io/milvus/issues/31495. 2024. https://github.com/milvus-io/milvus/issues/31495. Accessed: 2024-12-1.

[60] Uber. nilaway. 2024. https://github.com/uber-go/nilaway. Accessed: 2024-12-1.

[61] Go. nilness. 2024. https://pkg.go.dev/golang.org/x/tools/go/analysis/passes/nilness. Accessed: 2024-12-1.

[62] Optimism. ethereum-optimism/optimism/pull/9811. 2024. https://github.com/ethereum-optimism/optimism/pull/9811. Accessed: 2024-12-1.

[63] Bugaev L. buger/jsonparser/pull/204. 2024. https://github.com/buger/jsonparser/pull/204. Accessed: 2024-12-1.

[64] The Kubernetes Authors. Garbage Collection | Kubernetes. 2024. https://kubernetes.io/docs/concepts/architecture/garbage-collection/. Accessed: 2024-12-5.

[65] Milvus. milvus-io/milvus/issues/19263. 2024. https://github.com/milvus-io/milvus/issues/19263. Accessed: 2024-12-1.

[66] Milvus. milvus-io/milvus/issues/19715. 2024. https://github.com/milvus-io/milvus/issues/19715. Accessed: 2024-12-1.

[67] Wikipedia. Value numbering. 2024. https://en.wikipedia.org/wiki/Value_numbering. Accessed: 2024-12-1.

[68] GitHub. Static single-assignment form. 2025.
https://codeql.github.com/docs/codeql-language-guides/codeql-library-for-go/#static-single-assignment-fo rm. Accessed: 2025-04-26.

[69] Amazon Web Services. Potential Memory Leak Due to Slice Expression. 2024. https://github.com/aws/aws-sdk-go/issues/5327. Accessed: 2024-12-1.

[70] Rebert A, Kern C. Secure by Design: Google's Perspective on Memory Safety . Google Security Engineering, 2024.

[71] The MITRE Corporation. CWE-401: Missing Release of Memory after Effective Lifetime. 2024. https://cwe.mitre.org/data/definitions/401.html. Accessed: 2024-12-1.

[72] Cao S, Sun X, Bo L, et al. MVD: Memory-Related Vulnerability Detection Based on Flow-Sensitive Graph Neural Networks . ICSE '22: Proceedings of the 44th International Conference on Software Engineering. New York, NY, USA: Association for Computing Machinery, 2022: 1456-1468. DOI: 10.1145/3510003.3510219.

[73] Fan G, Wu R, Shi Q, et al. SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019: 72-82. DOI: 10.1109/ICSE.2019.00025.

[74] Sui Y, Ye D, Xue J. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. IEEE Transactions on Software Engineering, 2014, 40(2): 107-122. DOI: 10.1109/TSE.2014.2302311.

[75] Clang. Thread Safety Analysis . 2024. https://clang.llvm.org/docs/ThreadSafetyAnalysis.html. Accessed: 2024-12-1.

[76] Chen J, Yu D, Hu H. Towards an Understanding of Memory Leak Patterns: An Empirical Study in Python . Software Quality Journal, 2023, 31(4): 1303-1330. DOI: 10.1007/s11219-023-09641-5.

[77] Klabnik S, Nichols C. What Is Ownership? . 2024. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html. Accessed: 2024-12-1.

[78] Klabnik S, Nichols C. Validating References with Lifetimes . 2024. https://doc.rust-lang.org/stable/book/ch10-03-lifetime-syntax.html.

Accessed: 2024-12-1.

[79] Qin B, Chen Y, Yu Z, et al. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs . PLDI 2020: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2020: 763-779. DOI: 10.1145/3385412.3386036.

[80] Tu T, Liu X, Song L, et al. Understanding Real-World Concurrency Bugs in Go. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Providence RI USA: ACM, 2019: 865-878. DOI: 10.1145/3297858.3304069.

[81] Lange J, Ng N, Toninho B, et al. Fencing off go: Liveness and safety for channel-based programming. Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. Paris France: ACM, 2017: 748-761. DOI: 10.1145/3009837.3009847.

[82] Gabet J, Yoshida N. Static Race Detection and Mutex Safety and Liveness for Go Programs (extended version): arXiv:2004.12859. arXiv, 2021. DOI: 10.48550/arXiv.2004.12859.

[83] Liu Z, Zhu S, Qin B, et al. Automatically detecting and fixing concurrency bugs in go software systems. Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021. DOI: 10.1145/3445814.3446756.

[84] Liu Z, Xia S, Liang Y, et al. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering . Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Lausanne Switzerland: ACM, 2022: 888-902. DOI: 10.1145/3503222.3507753.

[85] Lauinger J, Baumgärtner L, Wickert A K, et al. Uncovering the Hidden Dangers: Finding Unsafe Go Code in the Wild. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). 2020: 410-417. DOI: 10.1109/TrustCom50675. 2020.00063.

[86] Saioc G V, Shirchenko D, Chabbi M. Unveiling and Vanquishing Goroutine Leaks in Enterprise Microservices: A Dynamic Analysis Approach. 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2024: 411-422. DOI: 10.1109/CGO57630.2024.104448 35.

[87] Gharat P, Shadab N, Tiwari S, et al. Resource Leak Checker (RLC#) for C# Code using CodeQL: arXiv:2312.01912. arXiv, 2023.

[88] Youn D, Lee S, Ryu S. Declarative Static Analysis for Multilingual Programs Using CodeQL . Software: Practice and Experience, 2023, 53(7):1472-1495. DOI: 10.1002/spe.3199.

[89] He D, Gui Y, Li W, et al. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. Proceedings of the ACM on Programming Languages, 2023, 7(OOPSLA2):256:971-256:1000. DOI: 10.1145/3622832.

李清伟(2001－),男,硕士研究生,主要研究领域为程序语言运行时与程序分析.

丁伯尧(1999－),男,博士研究生,主要研究领域为面向内存安全的程序分析、多语言程序交互与适配、现代语言编译和运行时系统.

张昱(1972－),女,博士,教授,CCF 杰出会员,主要研究领域为面向新兴计算的编程系统、软件分析与系统优化,如智能计算、数据计算、量子计算等.

陈金宝(1999－),男,博士研究生,主要研究领域为现代语言编译和运行时系统、软件安全.