软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(6):2585−2607 [doi: 10.13328/j.cnki.jos.007096]

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

DBI-Go: 动态插桩定位 Go 二进制的非法内存引用*

陈金宝, 张 昱, 李清伟, 丁伯尧

(中国科学技术大学 计算机科学与技术学院, 安徽 合肥 230026)

通信作者: 张昱, E-mail: yuzhang@ustc.edu.cn

摘 要: Go 语言, 也称 Golang, 由于其语法简单、原生支持并发、自动内存管理等特性, 近年受到很多开发者的欢迎. Go 语言期望开发者不必了解变量或对象是分配在栈上还是在堆中, 而由 Go 编译器的逃逸分析来决定分配位置, 再由 Go 垃圾收集器自动回收无用的堆对象. Go 的逃逸分析必须正确决定对象的分配位置以保证内存状态的正确性. 然而, 目前 Go 社区中逃逸相关问题频发, 潜在导致程序崩溃等致命问题, 而目前对该方面的研究缺失. 为有效检测编译器生成的代码是否存在可能引起运行时崩溃的非法内存引用, 填补研究空白, 对 Go 程序执行进行抽象建模, 并提出两条判定写入违例的规则. 基于这两条规则, 克服 Go 二进制中高层语义缺失、运行时信息不便获取等挑战, 设计一个轻量化的分析工具 DBI-Go. DBI-Go 采用静态分析加动态二进制插桩的分析方式, 基于动态二进制分析框架 Pin 来实现, 可以识别 Go 二进制中违例的 store 指令. 实验结果表明, DBI-Go 可以检测出 Go 社区中所有已知的逃逸相关 Issues; DBI-Go 还发现一个目前 Go 社区未知的问题, 该问题已经得到确认. 在实际项目上的应用则表明 DBI-Go 可以帮助开发人员找出逃逸算法的错误. 测试结果还表明 DBI-Go 采取的措施可以有效降低误报率且在 93.3% 的情况下带来的额外运行时开销小于原先的 2 倍. 同时, DBI-Go 无需修改 Go 的编译运行时, 可以适配不同版本的 Go, 有较高的适用性.

关键词: 二进制分析; 动态二进制插桩; 静态分析; Go; 编译器测试; 逃逸分析

中图法分类号: TP314

中文引用格式: 陈金宝, 张昱, 李清伟, 丁伯尧. DBI-Go: 动态插桩定位 Go 二进制的非法内存引用 . 软件学报, 2024, 35(6): 2585–2607. http://www.jos.org.cn/1000-9825/7096.htm

英文引用格式: Chen JB, Zhang Y, Li QW, Ding BY. DBI-Go: Dynamic Binary Instrumentation for Pinpointing Illegal Memory References in Go Binaries. Ruan Jian Xue Bao/Journal of Software, 2024, 35(6): 2585–2607 (in Chinese). http://www.jos.org.cn/1000-9825/7096.htm

DBI-Go: Dynamic Binary Instrumentation for Pinpointing Illegal Memory References in Go Binaries

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

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

Abstract: The Go programming language, also known as Golang, has become popular with developers in recent years due to its simple syntax, native support for concurrency, and automatic memory management. This language expects that developers do not need to know whether variables or objects are allocated on the stack or in the heap. The escape analysis of the Go compiler determines the allocation location, and then the garbage collector automatically recycles unreachable heap objects. Go's escape analysis must correctly determine the allocation location of the object to ensure the memory state correctness. However, escape analysis related problems frequently occur in the Go community at present, potentially causing fatal problems such as program crashes, and there is currently a lack of research on this

* 基金项目: 国家自然科学基金 (62272434)
本文由"编译技术与编译器设计"专题特约编辑冯晓兵研究员、郝丹教授、高耀清博士、左志强副教授推荐.
收稿时间: 2023-09-10; 修改时间: 2023-10-30; 采用时间: 2023-12-14; jos 在线出版时间: 2024-01-05
CNKI 网络首发时间: 2024-03-23

aspect. To effectively detect whether the code generated by the compiler has illegal memory references that may cause runtime crashes and fill the research gap, this study conducts abstract modeling on the Go program and proposes two rules for verifying the validity of store instructions. Based on these two rules, it overcomes the challenges of lacking high-level semantics in Go binaries and inconvenient access to runtime information and designs a lightweight analysis tool DBI-Go. DBI-Go adopts static analysis plus dynamic binary instrumentation and is implemented based on Pin, a dynamic binary analysis framework. Meanwhile, DBI-Go can identify illegal store instructions in Go binaries. Evaluation results show that DBI-Go can detect all known escape-related issues in the Go community, and also discover an issue that is previously unknown to the Go community. Finally, this issue has been confirmed. The applications in actual projects show that DBI-Go can assist developers in finding bugs in escape analysis algorithms. Evaluation results also show that the measures adopted by DBI-Go can reduce the false positive rate, and the extra runtime overhead brought by DBI-Go in 93.3% of the cases is less than twice the original. Additionally, DBI-Go can be adapted to different versions of Go without modifying Go's compilation and runtime, therefore yielding wide applicability.

Key words: binary analysis; dynamic binary instrumentation; static analysis; Go; compiler testing; escape analysis

Golang (简称 Go)[1]是由 Google 提出并于 2009 年开源的新兴编程语言. Go 语言因其语法简单、静态强类型、自动内存管理、原生支持高并发、编译型等特性, 得到许多开发者的认可. 2009 年和 2016 年两度成为 Tiobe 编程语言排行榜的年度明星[2]. 根据 2022 年 Go 开发者雇佣报告[3], 全球 10.5% 的开发人员将 Go 作为主力编程语言, 亚洲占比最高, 达 57 万人, 而中国更是有超过 16% 的开发者使用 Go 语言.

随着软件无处不在, 软件的安全及效能变得越来越重要, 而其中内存管理方式及其实现机制对软件开发产能、安全和效能的影响尤为重要. 传统的 C/C++ 语言需要开发者显式决定变量是分配在栈、静态数据区、还是堆中, 显式管理对象的分配与回收, 这使得程序极易出错, 给开发和维护带来极大的负担. 为提升内存安全性和软件开发产能, 越来越多的现代编程语言, 如 Java、Python、Go 等, 提供垃圾回收 (garbage collection, GC) 等自动内存管理机制, 使得开发者无需管理对象的回收. Python 采用以引用计数为主、以分代解决循环引用为辅的 GC[4], 其采用全局解释器锁 (GIL) 使多线程在竞争时串行执行. Java 虚拟机如 OpenJDK 则提供多种丰富的 GC 算法来专注于吞吐量 (throughput)、延迟 (latency)、或内存占用 (memory footprint) 等不同性能指标, 它们组合运用多线程 stop-the-world (STW, 多线程回收时让用户代码停止执行)、分代、紧致 (compaction, 回收期间通过移动对象来紧致内存以解决碎片问题) 或不同的并发 (concurrent, 回收与用户代码并发执行) 等技术[5]. 而 Go 的 GC 建立在基本没有碎片问题的分配器 TCMalloc[6]之上, 使用的是无分代的、无紧致、并发的三色标记清除算法, GC 算法轻量. 这些特性使得 Go 在特定场景下拥有 Python 般流畅的开发体验的同时, 又能达到接近 C++的运行效率[7].

Go 使用逃逸分析 (escape analysis) 在编译阶段来决定对象是否堆分配, 使得开发者无需手动指定对象的分配位置. 现有逃逸分析有关的研究主要集中在 Java 语言上[8−13], 而非 Go. 对于栈分配, 只需通过修改栈帧指针即可几乎零开销地完成分配和回收; 而堆分配则需要 GC 承担相当厚重的分配和回收代价. Java 语言呈现给开发者的观点是对象在堆中分配, 而具有局部作用域的基本类型的值和对象类型的引用才会在栈上分配. 为降低 GC 的负担, 现代 Java 虚拟机在即时编译器中利用逃逸分析结果来开展内存使用优化, 如将未逃逸出方法的对象进行标量替换 (即展开为一系列基本类型的值), 进而可以自动在栈上分配[10]. Go 语言则期望让开发者专注于程序功能逻辑本身, 不必指定变量或对象是分配在栈还是堆中, 而由编译器的逃逸分析来判定对象是否逃逸到堆上, 进而决定分配的位置. Go 语言的逃逸分析是编译流水线中的必要流程, 其还承担了维护 Go 内存安全以及 GC 正常运行的责任. Go 语言编译器的逃逸分析需要正确判断一个对象是否逃逸并分配在堆上, 否则极有可能产生若干内存漏洞, 如悬垂指针等, 导致 Go 程序运行时异常行为甚至崩溃 (panic), 在实际应用环境中造成较大损失.

一个正确的 Go 逃逸分析需要找到所有需要堆分配的解, 使得程序可以正常运行, 不会因为内存漏洞而崩溃. 然而, 目前 Go 社区中的逃逸问题频发, 据不完全统计, 从 2017 年至今已有 17 个和逃逸分析相关的 issues (如图 1 所示), 并且无减少收敛趋势 (如图 2 所示). 比如 2022 年的 issue#54247[14], 逃逸分析和后续编译优化的配合出错, 导致本该逃逸的对象栈分配, 造成堆对象引用了悬挂的栈指针; 2021 年的 issue#44614[15], 由于逃逸分析的分析出错, 导致全局对象引用了悬挂的栈指针. Go 的 GC 在运行时发现不正确的指针后会直接崩溃[16], 因此和逃逸相关的问题一旦出现, 通常都是致命的问题. 另外, 由于 GC 不是这种错误发生的第一现场, 因此通常难以定位这种运

行时崩溃产生的原因, 从而 Go 的 GC 的崩溃输出几乎不能提供有效的信息帮助开发者定位问题. 像前述的 issue#44614 从 Go1.14 便开始出现, 直至 Go1.17 发布前才被修复, 影响了 Go1.14.x、Go1.15.x、Go1.16.x 这 3 个大版本.

年份	issue 号	版本	备注
2022	54247	go1.17	堆指向栈的问题
2022	53289	go1.18	堆指向栈的问题
2022	51481	go1.18	堆指向栈的问题
2022	52008	go1.15	堆指向栈的问题
2021	47415	go1.16	CGO 相关逃逸问题
2021	49755	go1.16	堆指向栈的问题
2021	47276	go1.16	堆指向栈的问题
2021	44614	go1.14 至1.16	堆指向栈的问题
2021	43818	go1.17	寄存器传参问题
2020	41635	go1.16	逃逸调试信息错误
2020	42944	go1.16	堆指向栈的问题
2019	31573	go1.13	逃逸带来的 runtime 问题
2019	32959	go1.13	逃逸带来的编译问题
2018	29000	go1.11 至1.12	逃逸带来的 runtime 问题
2018	29353	go1.13	逃逸带来的 runtime 问题
2018	26987	go1.12	逃逸带来的 runtime 问题
2017	23045	go1.11	逃逸带来的 uintptr 问题

issue 链接: https://github.com/golang/go/issues/

图 1　社区中 Go 逃逸错误相关 issues (不完全统计)

图 2　社区中 Go 逃逸错误相关 issues 数量变化趋势

针对 Go 程序和/或 Go 语言编译器的漏洞检测, 目前多集中在对 Go 程序的并发、竞争检测方面[17−19], 与 Go 语言编译器的逃逸分析相关的研究屈指可数, 仅有 Wang 等人的工作[20]使得一些对象可以绕过 Go 语言编译器的逃逸分析, 从而节省堆内存的使用. 目前尚无相关工作来寻找经 Go 语言编译器编译出的代码中的错误内存引用, 学术界对此的研究缺失. 同时, 目前 Go 官方对逃逸分析正确性的测试手段也较为有限, 难以辅助开发人员对逃逸分析算法进行进一步的优化或重构.

为了有效检测编译器生成的代码是否存在可能引起运行时崩溃的非法内存引用, 在产品上线前就能及时发现问题, 避免产品在实际生产环境中出现崩溃, 造成损失, 也为了辅助开发人员对内存优化相关算法, 如逃逸分析进行优化和重构, 验证算法的正确性, 本文提出一种结合静态分析和动态插桩检测 Go 二进制中可能导致非法内存引用的 store 指令的方法, 并基于 Pin[21]实现了工具原型 DBI-Go, 它可以在不修改 Go 编译运行时系统的情况下对 Go 二进制程序进行检测, 具有较好的适用性. 该工具权衡静态分析和动态插桩, 并结合 Go 的语言特性, 大大减少了误报率和插桩带来的额外开销.

本文的主要贡献点包括以下内容.

• 对 Go 二进制程序进行抽象, 并基于此抽象分析提出两条判定非法内存引用的判定规则. 非法内存引用由二进制程序中将不当的地址通过 store 指令写入: (1) 将栈地址存入栈外; (2) 将较浅栈帧中的对象的地址存入较深栈帧.

• 提出了 DBI-Go——第 1 个使用动态二进制插桩方式分析 Go 二进制中潜在非法内存引用的工具. DBI- Go 可以将 Go 的二进制可执行文件与 Go 语义以及 Go 的运行时管理系统关联起来, 有着较低的误报率, 且在大多数情况 (93.3%) 有不超过 2 倍的额外运行时开销. DBI-Go 还可以精准给出违例的发生位置, 帮助快速定位问题.

• 实验结果表明, DBI-Go 可以检测出 Go 社区中所有已知的逃逸相关 issue, 有着较高的漏洞覆盖率. 同时 DBI-Go 还发现了一个目前 Go 社区未知的问题. 该问题已经在 golang-nuts 中得到 Go 官方维护人员的确认 (https://groups.google.com/g/golang-nuts/c/YZVFzwnPixM), 并已向社区提交 issue 有待 Go 官方的进一步修复 (https://github.com/golang/go/issues/61730).

• 对 DBI-Go 的实际应用还表明, 其可以辅助开发人员对 Go 逃逸分析算法进行优化和重构, 帮助找到新逃逸

分析算法的错误.

本文第 1 节介绍 Go 逃逸分析及现状, 引出研究动机. 第 2 节主要对 Go 二进制进行抽象并提出两条判定规则. 第 3 节介绍 DBI-Go 的设计挑战以及其最终的设计与实现, 包括其是如何恢复 Go 二进制上的运行时信息和语义信息及如何减少误报和降低开销. 第 4 节对 DBI-Go 的漏洞覆盖率、误报率以及带来的额外开销进行实验评估. 第 5 节对 DBI-Go 的局限性进行讨论. 第 6 节介绍相关工作. 第 7 节进行总结与展望.

1 相关基础与研究动机

本节简要介绍 Go 语言的运行时系统和逃逸分析, 并结合社区 issue 概述了目前 Go 逃逸分析的现状, 说明了目前对逃逸分析正确性验证的不足, 引出本文的研究动机.

1.1 Go 的运行时系统与逃逸不变式

1.1.1 Goroutine 及其栈管理

Go 语言使用协程 Goroutine[22] 作为 Go 程序的执行上下文. Goroutine 是轻量级的用户态线程, 与由操作系统直接调度的操作系统级线程 Thread 不同, Goroutine 的调度是由 Go 的运行时系统进行管理的. 每个 Goroutine 都有自己独有的栈, 但它的额外开销和默认栈大小都比线程小很多. 与操作系统线程的栈不同, Goroutine 的栈是 Go 运行时系统使用堆内存来模拟的. Go 运行时系统从操作系统申请堆内存后会长期持有, 再通过其内部的内存分配器按照一定的策略和时机从中划分出部分内存用于模拟 Goroutine 的栈.

图 3 示意了 Go 运行时系统对每个 Goroutine 的栈管理结构, 其中 stack 结构包含两个字段: lo 和 hi, 分别表示栈的低地址和高地址边界, 它们描述一个栈的内存地址范围位于 [lo, hi) 之间. 每个 Goroutine 在 Go 运行时系统中用一个 g 类型的对象表示, g 对象的前几个字段描述它的执行栈, 包括一个类型为 stack 的字段, 用于描述该 Goroutine 的栈的地址范围. Go 的运行时系统会在 Goroutine 的栈空间不足时进行栈扩展. 当发生栈扩展时, Go 运行时会进行栈拷贝, 将旧栈的内容复制到新分配的栈, lo 和 hi 也会进行相应的更改, 因此 Go 程序执行期间, 每个 Goroutine 的栈并非固定在内存中的同一段连续空间保持不变.

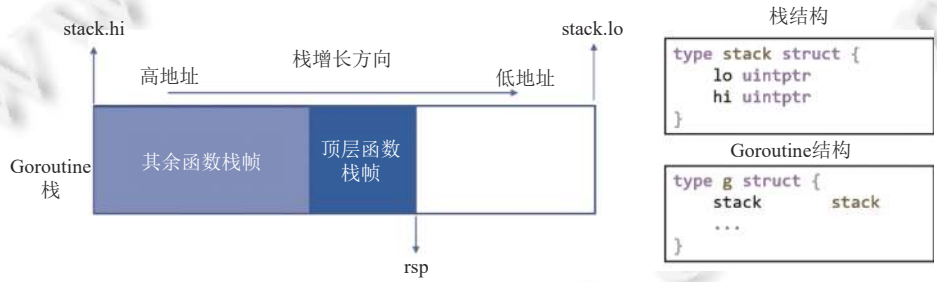


图 3 Goroutine 执行栈管理示意

1.1.2 Go 的垃圾回收与逃逸不变式

Go 语言内存管理主要依赖于其运行时系统, Go 从操作系统申请内存后会长期持有, 将其分为 Goroutine 栈 (如前文所述) 和 Go 堆进行管理. Go 堆使用 TCMalloc[6]进行快速的并发分配, 并通过 Go 的并发垃圾收集 (GC)[23] 实现堆空间的回收. Go 中的堆对象会使用诸如 runtime.newobject 等运行时函数在 Go 堆上自动分配.

这种内存管理机制使得 Go 程序员无需了解一个变量是分配在栈上还是堆中[24]. Go 向程序员保证, 程序执行的任何时刻中, 任意由垃圾回收标记算法标记可达的对象都处于生命周期内, 即, 不可能出现悬挂指针. 作为 Go 编译流水线上的一个重要优化遍, Go 的逃逸分析用于决定一个对象是堆分配还是栈分配. 为了内存安全以及 GC 的正常运行, Go 的设计者提出了以下两条逃逸不变式[25].

- 逃逸不变式 1: 指向栈对象的指针不可存储在堆中.
- 逃逸不变式 2: 指向栈对象的指针生命期不可超出该栈对象.

Go 的逃逸分析在决定对象是堆分配还是栈分配时必须遵循上述不变式. 逃逸分析若发现有对象违反上述不变式 (即违例), 则该对象会被堆分配. 如代码 1 所示, 其中 heapObj 是堆对象, 其在第 5 行引用了 i 的地址. 因此根据逃逸不变式 1: “指向栈对象的指针不可存储在堆中”, i 需要堆分配. 在代码 2 中, ptr 在第 6 行引用了 x 的地址. ptr 在外层循环声明, x 在内层循环隐式声明, ptr 的生命期长于 x. 因此根据逃逸不变式 2: “指向栈对象的指针生命期不可超出该栈对象”, x 需要堆分配.

代码 1. 逃逸不变式 1 示例.

```
1. //堆对象获取了 i 的指针导致 i 被堆分配
2. func heapAssignment() {
3.     heapObj := newobj()
4.     i := 1
5.     heapObj.a = & i
6.     use(heapObj)
7. }
```

代码 2. 逃逸不变式 2 示例.

```
1. //ptr 引用 x, ptr 循环深度小于 x, 生命期长于 x, x 堆分配
2. func loop() {
3.     var ptr *int
4.     for i := 0; i < 5; i++ {
5.         x := i
6.         ptr = & x
7.     }
8.     use(ptr)
9. }
```

逃逸分析必须正确地决定对象的分配位置, 若有对象的分配位置违反了上述两个逃逸不变式之一, 即存在非法内存引用, 则会导致 Go 程序运行时的异常行为甚至导致运行时崩溃 (panic), 在实际应用环境中造成较大损失.

1.2 Go 逃逸问题现状

1.2.1 Go 社区中逃逸类相关问题频发

在社区 issue 中, Go 的逃逸问题频发, 且每次出现的问题都是致命问题. 近几年出现的相关 issue 的不完全统计可见图 1. 这些 issues 有的是因为逃逸分析算法的错误导致, 有的是因为逃逸分析和其他编译优化的错误配合导致. 下面将通过这些 issue 中的两个典型例子来进行说明.

• 逃逸分析算法的错误. 逃逸分析算法的错误会直接导致一些对象的分配位置出错. 以 issue#44614 [15]为例, 其一个简化版的实例如代码 3 所示. 在该例子中, 对象 r 由于被全局变量 sink 引用而被堆分配. 函数 global2stack 中的参数 p, 在 return p 语句中被堆对象 r 获取, 因此所有传给 global2stack 的指针理应被标记为逃逸, 但由于逃逸分析的漏洞, 逃逸分析在分析 global2stack 函数时认为参数 p 没有逃逸. 这就导致了在代码 3 中, 地址被传给 global2stack 的变量 i 理应被堆分配, 但却被逃逸分析错误地认为应该栈分配.

代码 3. issue44614 简化版示例.

```
1. var sink interface{}
2. func global2stack(p *int) (r *int) {
3.     sink = & r
```

```
4.    return p
5. }
6. func g2s() {
7.    i := 1 //i 应该堆分配, 却被错误地栈分配
8.    j := global2stack(& i)
9.    _ = j
10. }
```

• 逃逸分析和后续编译优化的错误配合. 有时虽然逃逸分析没有出错, 但后续的一系列的编译优化却可能造成错误. 一个例子是社区 issue#54247 [14]. 其一个简化版的实例如代码 4 所示. 在该例中, obj1 是栈对象, 但逃逸分析后续编译流程中对 defer 的处理导致函数 Recover 的参数 objs 逃逸到堆上, 这就导致在第 4 行, 栈对象 obj1 的地址被堆对象 objs 获取, 违反逃逸不变式, 导致错误.

代码 4. issue54247 简化版示例.

```
1. func Escape(task func()) {
2.    var obj1 obj
3.    defer Recover(
4.       & obj1,
5.    ) //obj1 应该堆分配, 却被错误地栈分配
6.    task()
7. }
8. func Recover(objs ...*obj) {
9.       use(objs)
10. }
```

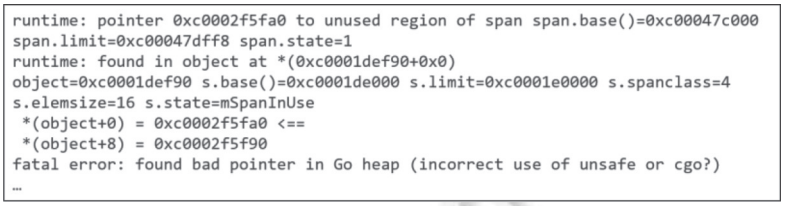
由于这些对象分配位置出错, 因此引用这些对象的堆和全局对象极易出现悬挂指针. Go 向程序员保证, 程序执行的任何时刻中, 任意可达的对象都处于生命周期内, 即, 不可能出现悬挂指针. 当 Go 的 GC 遇到悬挂指针时会直接在运行时 panic, 使得整个 Go 程序异常终止. 若在实际生产环境中出现该问题, 则很有可能带来不可挽回的损失. 同时, GC 不是发生这种错误的第一现场, 以代码 3 为例, 其发生错误的第一现场应是第 4 行的 return p, 在该处栈对象的地址被传递给堆对象. 因此, GC 的 panic 具有延后性和不可预知性. 由于 GC 不是错误的第一现场, 其崩溃输出多和 GC 相关而和事发的第一现场无关. 后文图 4 所示为 issue#44614 和 issue#54247 中的崩溃输出. 其多为显示 GC 的工作流程和状态信息, 不能准确地显示栈对象是在什么位置被传递给了全局或堆对象. 不完整的信息也给排查问题带来了困难, issue#44614 中逃逸分析的漏洞从 Go1.14 开始出现, 直至 Go1.17 发布前才被修复, 影响了 Go1.14 至 Go1.16 这 3 个大版本.

1.2.2 逃逸分析正确性验证的现状

Go 语言官方对逃逸分析正确性的验证手段也较为有限, 目前在 Go 语言官方给出的逃逸分析的测试中对逃逸分析正确性的检验存在比较函数返回值和比较 DeBug 信息这两种途径.

途径一[26]是通过检查两次调用同一个函数的返回值是否相同来判断逃逸分析正确性的. 在这个函数内部会为一个对象分配一块空间, 这部分空间理应堆分配, 这样在两次函数调用中将该空间的地址返回时为不同的地址, 这就可以通过测试. 但是如果该空间由于错误的逃逸分析导致被栈分配, 因为栈帧的布局在编译完成后已经确定, 故将导致在同一个栈帧下两次调用该函数返回的该变量的地址将为相同的地址, 这样就不能通过测试. 这种情况只能适用于小规模测例, 而且要求在同一个栈帧下调用相同函数进行测试, 较难推广.

(a) issue44614 中的崩溃输出



(b) issue54247 中的崩溃输出

图 4 崩溃输出示例

另外一种途径[27]是通过标注的方式来进行检查的. 通过将预先人工标注的信息以及逃逸分析在 debug 模式下打印输出的信息进行比对来验证逃逸分析算法的正确性.

这两种途径从本质上来讲都是通过标注测试的方式来实现的, 需要事先人工分析 Go 程序并进行标注, 其无法用于验证实际应用程序中事先未知的错误.

此外, Go 的运行时系统在 GC 时会验证对象的指向是否有错, 但该方法的范围有限. 首先, 并不是所有错误的指向都能被 GC 发现. 其次, 该验证机制将错误的指向延迟到 GC 时才进行检测, 不能及时发现问题, 且 GC 一旦发现问题会直接 panic, 造成整个程序的崩溃, 可能在实际生产环境中造成不可弥补的损失. 最后, GC 的崩溃输出较难理解, 难以帮助开发人员定位问题的发生位置, 为后续排查工作带来不便.

总之, Go 目前测试和验证方法十分有限, 在实际应用程序中发生错误时, 现有方法无法用于定位发生错误的位置以及发生错误的原因. 图 1 中的各类 issue 也印证了 Go 现有方法的局限性.

1.3 研究动机

传统的内存相关漏洞的寻找工作多集中于 C/C++程序的二进制[28−30]. 但由于 Go 程序有独特的运行时管理机制以及独特的语义, 传统基于 C/C++的工作并不能直接应用在 Go 二进制上. 目前学术界在 Go 程序的漏洞寻找上有不少工作, 但这些工作多集中在并发、数据竞争相关的漏洞寻找上[17,18,31]. 目前缺少验证 Go 编译器生成的代码是否满足 Go 逃逸不变式的研究.

如何保证 Go 编译器中逃逸分析的决策结果以及经过其他编译器优化遍之后生成的代码仍然符合逃逸不变式是非常重要的. 然而, 正如第 1.2 节所述, Go 目前逃逸相关 issue 频发且无较为有效的测试手段, 且现有相关研究缺失. 为了有效解决这类问题, 确保经过逃逸分析和一系列优化遍后生成的二进制可执行文件中没有违反 Go 逃逸不变式的情况, 研制一个可以有效检测实际应用中违反 Go 逃逸不变式情况的工具是很有必要的.

2 Go 程序逃逸不变式违例的判定规则

本节对 Go 程序进行抽象, 随后结合 Go 的逃逸不变式抽象出 Go 程序中违反逃逸不变式的判定规则.

2.1 Go 程序抽象

根据 Go 的文档[23,32], Go 应用程序的内存由全局数据区、受 GC 管理的 Go 堆区、每个 Goroutine 的栈区组成. 基于此, 我们将一个 Go 程序的内存分为 3 部分: Goroutine 栈、Go 堆区以及 Go 全局数据区, 并提出如公式 (1) 所示的抽象. 整个世界 $\mathbb{W}$ 由一个含有 n 个 Goroutine 的集合 $\mathbb{GS}$、一个 Go 堆区 $\mathbb{GH}$、一个 Go 全局数据区 $\mathbb{GG}$ 组成. 每个 Goroutine $\mathbb{G}$ 由运行在其上的用户代码 $\mathbb{C}$ 以及对应的 Goroutine 栈 $\mathbb{S}$ 组成. $\mathbb{GH}$、$\mathbb{GG}$、$\mathbb{S}$ 均是内存地址的集合, 其中: $\mathbb{GH}$ 和 $\mathbb{GG}$ 分别记录 Go 程序在用的有效堆地址和全局数据区地址; $\mathbb{S}$ 是 Goroutine 的栈地址, 由从 $addr_{lo}$ 到 $addr_{hi}$ 的连续内存地址组成. $\mathbb{C}$ 是一个指令列表, 由于这里只关心潜在引起 Go 逃逸不变式违例相关的指

令, 因此指令列表中只包含涉及存储指针的 store 指令和可能引起控制流变化的 cmp 和 br 指令. 其中指令 store addr_{dst}, addr 代表将 addr 存入 addr_{dst} 所指向的内存区域.

$$
\begin{array}{lllll}
\text{(World)} & \mathbb{W} & ::= & (\mathbb{GS},\mathbb{GH},\mathbb{GG}) \\
\text{(Goroutine set)} & \mathbb{GS} & ::= & (\mathbb{G}_1,\mathbb{G}_2,\ldots,\mathbb{G}_n) \\
\text{(Go heap)} & \mathbb{GH} & ::= & \{\text{addr}\} \\
\text{(Go global)} & \mathbb{GG} & ::= & \{\text{addr}\} \\
\text{(Goroutine)} & \mathbb{G} & ::= & (\mathbb{C},\mathbb{S}) \\
\text{(Code)} & \mathbb{C} & ::= & [\text{instr}] \\
\text{(Goroutine stack)} & \mathbb{S} & ::= & [\text{addr}_{\text{lo}},\text{addr}_{\text{hi}}) \\
\text{(Memory address)} & \text{addr} & ::= & \text{n (unsigned nums)} \\
\text{(Instruction)} & \text{instr} & ::= & \text{store addr}_{\text{dst}}\text{, addr} \\
 & & & \mid \text{br addr} \mid \text{cmp}
\end{array} \tag{1}
$$

$$
\begin{array}{rll}
\mathit{code}(\text{ instr }) & \overset{\text{def}}{=} \mathbb{C} & \text{instr} \in \mathbb{C} \\
\mathit{codeG}(\mathbb{C}_0) & \overset{\text{def}}{=} \mathbb{G} & \mathbb{G}\cdot\mathbb{C} == \mathbb{C}_0 \\
\mathit{codeS}(\mathbb{C}) & \overset{\text{def}}{=} \mathbb{S}_0 & \mathbb{S}_0 == \mathit{codeG}(\mathbb{C}).\mathbb{S}
\end{array} \tag{2}
$$

2.2 违反 Go 逃逸不变式的判定规则

为便于描述, 首先引入一些辅助函数, 定义见公式 (2). 其中 *code*(instr) 用于获取指令 instr 所在的指令序列 $\mathbb{C}$, *codeG*($\mathbb{C}$) 用于获取执行指令序列 $\mathbb{C}$ 的 Goroutine, *codeS*($\mathbb{C}$) 用于获取执行指令序列 $\mathbb{C}$ 的 Goroutine 栈.

为了后面讨论违反逃逸不变式的判定规则以及栈对象的生命期, 接下来定义两个概念.

定义 1 (不合法的 store 指令). 若一条 store 指令 s_l: store addr_{dst}, addr 的内存访问违反了 Go 逃逸不变式, 则将其记为 $\mathit{illegal}(s_l)$.

定义 2 (栈对象所在的栈帧深度). 若 addr 为某栈对象地址, 则 $fd(\text{addr})$ 代表 addr 指向的栈对象所在的栈帧深度. 处于栈顶的函数栈帧深度为 1, 其余函数的栈帧深度沿着栈上的调用链依次加 1. 越靠近栈顶的栈帧, 其栈帧深度越小.

根据第 1.1 节所述的两条 Go 逃逸不变式, 可得违反 Go 逃逸不变式的情况为:

• 违反逃逸不变式 1: 栈对象指针被堆对象获取.

• 违反逃逸不变式 2: 栈对象指针被生命期更长的对象获取.

针对违反逃逸不变式 1 的情况, 其表现为堆对象指向了栈对象, 栈对象地址在某处被存入堆对象中. 因此, 针对指令 s_l: store addr_{dst}, addr , 可用公式 (3) 来判断是否违反了逃逸不变式 1, 即 addr_{dst} 是堆地址且 addr 是当前执行指令 s_l 所在的 Goroutine 栈中的地址时, 指令 s_l 会因引起逃逸不变式的违例而视为不合法.

$$
\frac{\mathbb{S} = \mathit{codeS}(\mathit{code}(s_l)) \qquad \text{addr} \in \mathbb{S} \wedge \text{addr}_{\text{dst}} \in \mathbb{GH}}{\mathit{illegal}(s_l)} \tag{3}
$$

针对违反逃逸不变式 2 的情况, 比当前栈对象生命期更长的对象可能有多种情况, 包括堆对象、全局对象、其他 Goroutine 的栈对象, 以及当前 Goroutine 的栈对象. 前 3 种情况都认为是当前 Goroutine 栈之外的对象. 下面分别进行讨论.

• 将栈对象地址写入当前 Goroutine 栈之外的违例情况. 除了公式 (3) 讨论过的堆对象外, 所有全局对象都可认为比当前栈对象生命期更长. 此时表现为全局对象指向了栈对象, 栈对象地址在某处被存入全局变量中. 因此, 针对指令 s_l: store addr_{dst}, addr, 可用公式 (4) 来判断是否将栈地址存入全局变量中.

$$
\frac{\mathbb{S} = \mathit{codeS}(\mathit{code}(s_l)) \qquad \text{addr} \in \mathbb{S} \wedge \text{addr}_{\text{dst}} \in \mathbb{GG}}{\mathit{illegal}(s_l)} \tag{4}
$$

由于不同 Goroutine 的执行受 Go 运行调度的影响可能相互交叠, 分处在不同 Goroutine 栈中的栈对象生命期可能存在交叠, 也可能存在不确定的一先一后, 因此将当前 Goroutine 中的一个栈对象的地址存到另一个 Goroutine 栈中也是不安全的. 为此, 针对指令 s_l: store addr_{dst}, addr, 可用公式 (5) 来判断是否将当前 Goroutine 中的栈地址存入了另一个 Goroutine 栈中.

$$\frac{\mathbb{S}=codeS(code(s_l)),\mathbb{G}_1=codeG(code(s_l)) \qquad \exists\mathbb{G}_2,\ \text{addr}\in\mathbb{S}\wedge\text{addr}_{\text{dst}}\in\mathbb{G}_2.\mathbb{S}\wedge\mathbb{G}_1\neq\mathbb{G}_2}{illegal(s_l)} \tag{5}$$

由于 Go 的地址空间由 Go 堆、全局数据区, 以及所有 Goroutine 的栈组成, 因此, 一个对象若不在当前 Goroutine 栈中, 则要么在 Go 堆中, 要么在全局数据区, 要么在其他 Goroutine 栈中. 即:

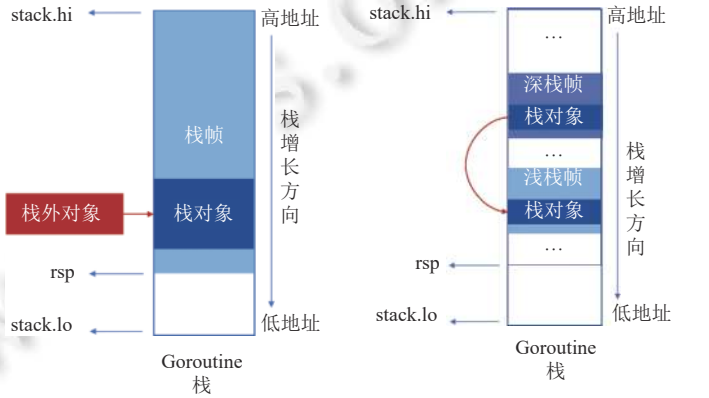
$$\begin{cases}\mathbb{S}=codeS(code(s_l)),\mathbb{G}_1=codeG(code(s_l))\\ \text{addr}_{\text{dst}}\notin\mathbb{S}\iff\text{addr}_{\text{dst}}\in\mathbb{GH}\vee\text{addr}_{\text{dst}}\in\mathbb{GG}\vee(\exists\mathbb{G}_2,\ \text{addr}_{\text{dst}}\in\mathbb{G}_2.\mathbb{S}\wedge\mathbb{G}_1\neq\mathbb{G}_2)\end{cases} \tag{6}$$

结合公式 (6), 可将公式 (3)、公式 (4)、公式 (5) 合为一个式子, 此时便得到判定违反逃逸不变式的第 1 条规则, 该规则揭示了栈对象地址被写入当前 Goroutine 栈之外的内存使用违例情况.

规则 1. 栈对象地址被写入当前 Goroutine 栈之外的违例判别.

$$\frac{s_l:store\ \text{addr}_{\text{dst}},\text{addr} \qquad \mathbb{S}=codeS(code(s_l)) \qquad \text{addr}\in\mathbb{S}\wedge\text{addr}_{\text{dst}}\notin\mathbb{S}}{illegal(s_l)}.$$

该规则表示将当前 Goroutine 栈的对象地址存入当前 Goroutine 栈外, 导致栈外对象引用栈内对象是不符合 Go 的逃逸不变式要求的, 如图 5(a) 所示.



(a) 栈外对象指向栈对象 (b) 较深栈帧栈对象指向较浅栈帧栈对象

图 5 规则 1 及规则 2 示意

• 将栈对象地址写入当前 Goroutine 栈内的违例情况. 在同一个 Goroutine 栈中, 不同栈对象生命期亦有差距, 比如作用域较浅的栈对象比作用域深的栈对象生命期长, 较深函数栈帧中的栈对象比较浅函数栈帧中的栈对象生命期长. 由于在二进制中已经难以看到源代码层级的作用域, 此处只能通过栈对象所在函数栈帧的深浅来判断其生命期, 认为较深栈帧栈对象生命期更长, 如图 5(b) 所示. 由此可以引出判定违反逃逸不变式的第 2 条判定规则, 即规则 2.

规则 2. 栈对象地址被写入当前 Goroutine 栈内的违例判别.

$$\frac{s_l:store\ \text{addr}_{\text{dst}},\text{addr} \qquad \mathbb{S}=codeS(code(s_l)) \qquad \text{addr}\in\mathbb{S}\wedge\text{addr}_{\text{dst}}\in\mathbb{S}\wedge fd(\text{addr})<fd(\text{addr}_{\text{dst}})}{illegal(s_l)}.$$

3 DBI-Go 的设计与实现

本节将介绍 DBI-Go, 一款用于识别 Go 二进制中写入指针的 store 指令并在运行时验证其是否违反 Go 逃逸不变式的工具的具体设计与实现.

3.1 设计目标和设计思路

DBI-Go 的设计目标主要包括以下几点.

• 轻量快速. 该工具应该尽可能轻量化, 可以以较快的速度分析出结果, 提升效率.

• 适用性强. 该工具应该尽可能独立于 Go 的编译工具链, 可以支持不同 Go 版本编译器生成的二进制, 支持在未修改编译运行时的原生 Go 二进制上进行检测.

• 低误漏报. 有效降低误报率可以大大减少人工筛选漏洞的时间. 同时, 减少漏报率可以确保该工具可以有效发现潜在的漏洞.

为达到上述设计目标, 对 DBI-Go 的设计的主要思路是结合 Go 语言的特性来快速原型化. 整个设计中的关键主要聚集于两点: (1) 程序分析方式的选择; (2) 如何结合并利用第 2 节抽象并总结出的违例判定规则.

• 分析方式的选择. 目前学术界已经开发了多种工具[17,18,33,34] 来识别 Go 应用程序中的各类漏洞, 主要使用两种分析技术——静态分析和动态分析. 静态分析无需执行即可分析 Go 源代码. 然而, 静态分析在识别方面的覆盖范围有限. 此外, 由于不精确的指针分析, 静态分析可能会引入许多误报或漏报. 当在实际的 Go 应用程序中进行大范围分析时, 这种不精确性会迅速累积. 因此, 单纯的静态分析难以满足 DBI-Go 的低误漏报的设计目标.

现有的动态分析通过额外的运行时信息监视 Go 程序的执行, 可以减少误报和漏报. 但是这些动态方法需要修改 Go 编译器或运行时来收集必要的数据, 与 Go 的编译运行时强耦合. 当 Go 的编译运行时发生改变时, 这些动态方法很容易失效. 这种要求很大程度上阻碍了这些工具在实际生产环境中的 Go 应用上的使用, 也不符合 DBI-Go 的适用性强的设计目标.

在 Go 应用程序的二进制代码上进行动态分析可以摆脱这些问题, 既可以在运行时监视 Go 程序的执行, 又无需修改 Go 的编译运行时. 因此, DBI-Go 以动态二进制插桩作为主要的分析方式.

• 如何利用规则 1 和规则 2. 确定了程序分析方式之后, 就可以着手设计 DBI-Go. 第 2 节抽象总结的规则 1 和规则 2 仅提供了判定的理论. 若要使用规则 1 和规则 2, 通过观察其形式, 不难发现, 在二进制代码上使用该规则必须考虑以下两个关键问题.

(1) 如何识别存储指针的指令 store $addr_{dst}$, addr: 这类指令改变内存之间的引用关系, 潜在可能违背逃逸不变式.

(2) 如何获取 Go 运行时 Goroutine 的栈信息: 由第 1.1.1 节可知, Go 程序执行期间 Goroutine 的栈地址区间不是一成不变的, 因此需要有途径准确获取当前时刻的栈信息.

这两个关键点也是程序分析的难点. 第 3.2 节将介绍使用静态分析识别写入指针的 store 指令的挑战以及我们的解决方案. 第 3.3 节将介绍如何从较为封闭的 Go 运行时中获取 Go 程序运行时 Goroutine 的栈信息.

DBI-Go 的整体架构如图 6 所示. DBI-Go 主要由两个组件组成.

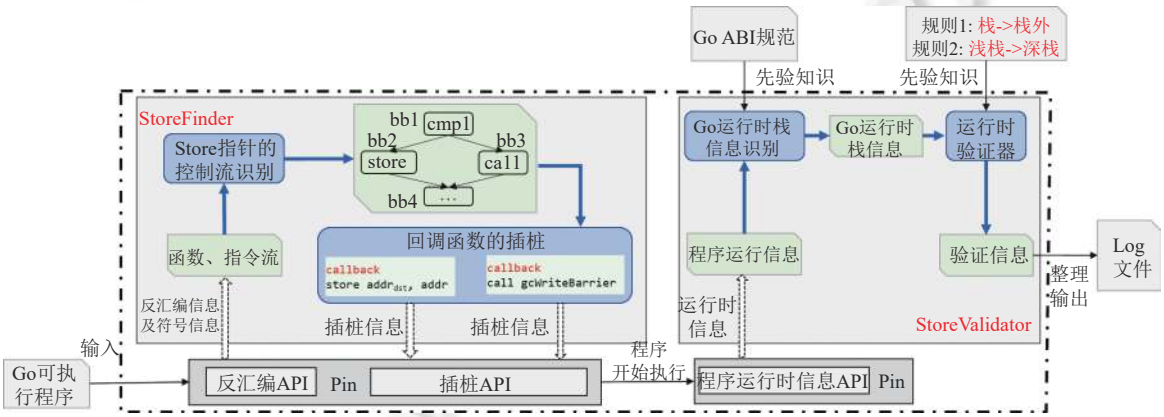


图 6 DBI-Go 整体架构

(1) *StoreFinder*: 它使用静态分析提取 Go 二进制程序中存入指针的 store 指令, 并在二进制代码上插桩 (第 3.2 节).

(2) *StoreValidator*: 它以运行时回调函数的形式识别违反 Go 逃逸不变式的内存引用漏洞, 并输出错误日志信

息, 便于开发者定位错误 (第 3.3 节).

DBI-Go 基于 Pin[21] 设计实现了上述两个组件, 包括约 1 000 行 C++代码. Pin 是由 Intel 开发的支持 IA-32、X86-64 和 MIC 指令集架构的动态二进制插桩框架, 可用于实现动态程序分析工具.

3.2 使用静态分析从 Go 二进制中恢复 Go 的 store 指针语义

Go 是编译型语言, Go 程序一旦被编译链接后其二进制代码中所有指令都会固定下来. 因此, 可以通过静态分析的方式识别 Go 程序二进制代码中的 store 指令. 然而, 若要判定一条 store 指令是否在存储指针 (即规则 1 和规则 2 中的 store $addr_{dst}$, addr 形式), 就还需要获取 Go 程序的一些高层语义信息, 如类型等. 下面先分析难点, 然后给出解决思路及其关键点的实现方法.

3.2.1 难点分析及解决思路

根据 Zeng[35] 的工作, 在 C 语言的二进制中, 所有高级类型信息, 如整型、浮点型和指针类型, 在编译后都会丢失, 二进制代码中仅有的两种类型是寄存器和内存位置. 在 Go 语言中也类似, Go 二进制程序已经难以区分指针类型和非指针类型.

• Go 二进制中指针的识别难点. Go 语言中, 指针类型用于传递对象地址, 不能进行指针运算. Go 的 GC 会扫描指针, 堆指针指向的对象会在合适的时机被 GC 回收. 然而, Go 语言中有一种类型 uintptr[36], 其大小和普通指针相同, 可以容纳任意的指针类型的值, 可以用于进行指针运算等操作. 但 GC 并不把 uintptr 当作指针, 因此也不会基于该指针值进行对象标记. 若把某栈对象的指针转为 uintptr 后存入堆对象, 并在之后不通过它访问对象, 那么这种存入操作并不违反 Go 的逃逸不变式, 因 Go 并不将 uintptr 视作指针. 然而, Go 二进制中仅有的两种类型是寄存器和内存位置, 难以判断某个寄存器或者内存位置中的值是指针还是 uintptr. 若对其不做区分, 都视为指针, 则势必会带来很多误报, 影响精度和效率.

• Go 二进制中地址存入操作的识别难点. 除了类型信息的缺失, Go 二进制上的地址存入操作与用户代码中的地址存入操作也有较大差别. Go 编译器在编译 Go 程序时会执行若干程序变换, 在用户代码中生成诸多与 Go 运行时管理相关的代码, 以便 Go 运行时系统对 Go 程序的管理. 在这些代码中会产生若干违反 Go 逃逸不变式 store 操作, 但 Go 的运行时保证了这些操作的安全性. 若不将这些操作排除, 也会带来较多的误报.

• 解决思路. Zhong 等人[17] 通过 Go 运行时系统中管理并发的相关函数恢复了 Go 二进制上的并发语义. 这启发我们可以通过静态分析的方式, 识别 Go 二进制用户代码中和内存管理、垃圾回收相关的 Go 运行时函数来恢复相关的 store 指针的语义. 经过对 Go 运行时相关函数的分析可发现, Go 运行时和写屏障相关的运行时函数可以用来恢复相应的 store 指令. 第 3.2.2 节中介绍该机制, 并在第 3.2.3 节中介绍如何使用该机制来恢复满足要求的 store 指令并使用 Pin API 为其注册运行时的回调函数.

3.2.2 Go 的写屏障

Go 运行时的不可或缺的部分为垃圾回收 (GC) 系统. 尽管 GC 在幕后运作, 却有数个运行时函数与其息息相关. 这些运行时函数分为两类: 一部分可供用户主动调用, 用于配置 GC 参数或强制启动新的 GC 周期; 另一部分由编译器在编译期间自动插入, 在运行时辅助 GC 的运行, 确保 GC 相关内存状态的准确性. 在这个体系中, 编译器自动插入的 runtime.gcWriteBarrier 函数在维护 GC 相关内存状态的正确性方面扮演着关键角色.

Go 的 GC 系统在堆内存使用达到特定阈值时会中断用户程序的运行, 对那些由根集中的指针直接或间接可达的对象进行扫描和标记, 标记出仍在生命周期中的对象, 随后释放已经不再使用的对象. 在这个过程中, 用户程序完全暂停, 因此在逻辑上对 GC 的发生没有感知, 这确保了内存读写不会对 GC 状态造成任何干扰.

Go 的并发 GC 在特定阶段允许用户程序与其并行, 以减少等待时间. 然而, 在 GC 对对象进行标记的过程中, 用户程序的内存读写可能会修改对象的引用关系, 这可能导致 GC 的标记与实际情况不一致, 从而错误地清理正在使用的对象. 例如, 如果在 GC 完成对栈指针的扫描后, Go 应用程序将某个堆对象的地址存入栈对象中, 而这个新引用关系的创建没有被 GC 感知到, 即该堆对象没有被 GC 标记, 那么在这一轮 GC 结束后, 该栈对象持有了一个已经被释放的堆对象地址, 从而导致内存错误. 因此, 在 GC 运行期间, GC 需要获知所有指针类型的内存写入,

以检查这些在运行过程中发生改变的引用关系. 在 Go 程序中, 如果一个 store 操作可能存入指针类型, 则 Go 编译器会在编译期间在该 store 操作周围生成特定的控制流, 插入 runtime.gcWriteBarrier 函数, 该函数在 GC 时会接管相应的 store 操作, 并负责帮助 GC 维护正确的内存引用关系.

Go 编译器只会对用户代码中包含指针的 store 操作插入 runtime.gcWriteBarrier, 且不会对一些可能带来误报的运行时函数中的 store 操作插入该函数. 若能够在二进制中识别相应的结构, 就能恢复 Go 的 store 语义, 大大减轻由于 Go 运行时、非指针 store 等带来的误报问题, 同时还能够减少很多对不必要的 store 插桩, 减轻动态二进制插桩带来的额外开销.

3.2.3 利用 Go 的写屏障机制恢复 store 指针的语义

编译生成的与 runtime.gcWriteBarrier 相关的汇编级控制流模式如图 7 所示. 由于编译器在编译时在 store 周围插入的特定控制流有固定的结构, 所以其最后生成的二进制中围绕 runtime.gcWriteBarrier 也有特定的结构. 将这些特征总结抽象, 可以形成图 7 中的特定控制流模式, 以便后续精准识别.

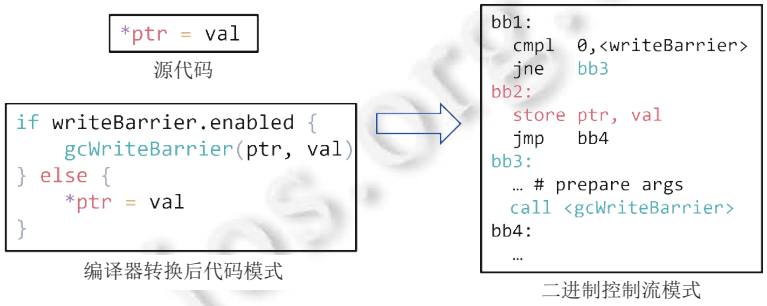


图 7 编译生成的 gcWriteBarrier 相关控制流

我们提出算法 1 来识别 Go 二进制中该特定的控制流模式. 其主要思路在于通过寻找特定的 cmpl 指令来确定满足要求的基本块 bb_1, 再通过 bb_1 中的控制流跳转语句 (JNE) 来确定满足要求的两个后继基本块 bb_2 和 bb_3, 要求 bb_2 和 bb_3 有且仅有一个相同的后继.

算法 1. 识别图 7 中的控制流并为其中满足 store 指针语义的指令设置回调函数.

输入: Go 二进制文件 *Bin*;

输入: *Bin* 中全局变量 runtime.writeBarrier 的地址 *wb*.

运行结果: 识别出 *Bin* 中满足 Go 的 store 语义的指令, 并为其在 Pin 中注册运行时的回调函数

1. **FOR ALL** *instr* in *Bin* **DO**
2. **IF** *instr.opcode*==cmpl **THEN**
3. **IF** *instr.operands*[0]==0 且 *instr.operands*[1] 的有效地址 == *wb* **THEN**
4. *j*=*instr* 之后最近的 jne 指令, 且该 jne 之前没有其他跳转指令
5. bb_3=*BB*(*j.target*) *j* 的目的地址处的基本块
6. bb_2= *BB*(*j.next*) *j* 的下一条指令处的基本块
7. **IF** *len*(*successors*(bb_2))==1 且 *successors*(bb_2)==*successors*(bb_3) **THEN**
8. **FOR ALL** *s*:store in bb_2 **DO**
9. 在 store 指令 *s* 前注册运行时回调函数
10. **END FOR**
11. **FOR ALL** *c*:call in bb_3 **DO**
12. **IF** *c.targetFunc* 以 runtime.gcWriteBarrier 为前缀 **THEN**

```
13.                识别 call 指令 c 的参数
14.                在 call 指令 c 前注册运行时回调函数
15.              END IF
16.            END FOR
17.          END IF
18.        END IF
19.      END IF
20. ENDFOR
```

在算法 1 中, 基本块 bb_2 中的所有 store 指令被认为都可能存储指针, 并将这些 store 转为规则 1 和规则 2 接受的形式: store $addr_{dst}$, addr. 之后, 通过使用 Pin 的 API 为这些 store 指令注册运行时回调函数. 在程序恰好运行到这些 store 指令之前时, 注册的回调函数会被执行用来检测其是否违反了 Go 逃逸不变式.

对于基本块 bb_3, 它对应在 GC 期间调用 runtime.gcWriteBarrier 函数来接管 store. 因此, 可以在 bb_3 中识别 runtime.gcWriteBarrier 所需的参数 (即图 7 中所示的 ptr 和 val). 在实现中发现, Go 编译器为了优化调用 runtime.gcWriteBarrier 的流程, 减少准备参数的开销, 为 runtime.gcWriteBarrier 函数生成了不同版本, 这些不同的版本只有传参的寄存器有区别. 比如 runtime.gcWriteBarrierR9 函数, 相比于原版的 runtime.gcWriteBarrier, 参数 val 使用寄存器 R9 来进行传递, 其余流程均与 runtime.gcW-riteBarrier 相同. 为此, 可以识别 runtime.gcWriteBarrierRXX 的后缀来判断其传递 val 参数的寄存器. 当识别出 runtime.gcWriteBarrier 的参数后, 就可将其转为 store $addr_{dst}$, addr 的形式 (ptr 对应 $addr_{dst}$, val 对应 addr), 并在 call 指令前注册运行时的回调函数用于在运行时检测该 call 指令所代表的 store 是否违反了 Go 逃逸不变式.

3.3 在运行时回调函数中恢复 Go 运行时栈信息

Go 运行时函数库以静态链接的方式与 Go 应用代码链接起来形成可执行的 Go 程序. Go 运行时函数负责在运行时管理 Go 程序运行所需的堆、Goroutine 的调度以及 Goroutine 的栈等. 用户编写代码时无需了解运行时的实现细节, 比如对象如何分配, Goroutine 如何调度, Goroutine 栈如何管理等. 但是, 如若要在二进制层面分析内存的引用关系, 分析栈对象地址是否被存储到栈外、是否违反逃逸不变式, 这就要求必须能够获得受 Go 运行时管理的一些信息, 比如当前 Goroutine 的栈信息等. 然而, Go 的运行时管理系统并不像操作系统一样提供了若干 API 用于在外部获取系统运行时信息. Go 的运行时系统相对封闭, 没有完备的 API 供外部获得当前 Go 程序的运行时信息.

幸运的是, 我们注意到 Go 语言的 ABI 规范[32]中定义了一些运行时信息的存储位置, 比如当前的 Goroutine, 来供运行时函数使用. 这意味着可以通过在二进制中添加回调函数的方式获得这些运行时信息.

在第 3.2.3 节中, 已为满足要求的 store 指令注册了运行时的回调函数. 该回调函数需要结合运行时信息来使用规则 1 和规则 2 检测这些 store 是否违反了 Go 的逃逸不变式. 第 3.3.1 节将介绍该回调函数在运行时如何利用 Go 的 ABI 从 Go 二进制中获得当前执行指令的 Goroutine 及其栈的相关信息.

3.3.1 在运行时回调函数中获得 Goroutine 栈信息

由第 1.1 节可知, Goroutine 栈是在操作系统进程的堆内存中模拟, 那么要获知 Goroutine 栈的范围, 判断某个指针是否是栈指针就不能简单地用操作系统的系统栈去判定. 为了得到 Goroutine 的栈信息, 需要获得运行时中用于管理 Goroutine 的 g 对象. 之后通过解析 g 对象的前几个字段的内存布局即可获得相应 Goroutine 的栈信息.

虽然 Go 运行时中的 g 对象不对用户暴露, 但幸运的是, 根据 Go 的 ABI-Internal[32]的约定可知, 在 AMD64 架构中, R14 寄存器会保存当前执行的代码所在的 Goroutine, 也就是 g 对象的地址. 再结合 ABI-Internal 中有关基本类型大小和对齐的约定以及前文所述的 g 和 stack 的结构, 可以通过公式 (7) 获得当前栈的 lo、hi:

$$\begin{cases} \text{lo} = *REG_{R14} \\ \text{hi} = *(REG_{R14} + 8) \end{cases} \tag{7}$$

在利用 Go 的 ABI-Internal 获得 Go 的运行时栈信息时, 我们发现旧版本的 Go (Go1.16.15 及以下) 的 ABI 与较新版本 Go (Go1.16.15 以上) 现行的 ABI-Internal 不同. 旧版本 Go 中, g 对象的地址不在寄存器 R14 中, 且旧版本的 Go 并没有相应的 ABI 文档. 为了了解如何获得旧版本 Go 中的运行时信息, 我们对旧版本的 Go 的编译运行时系统进行了人工分析. 最终发现在旧版本 Go 中, g 对象的地址存放在 TLS (thread local storage) 中的固定位置, 作为线程本地存储的一部分. 为了区分新版本和老版本的 Go, DBI-Go 在加载二进制时, 会首先获得系统 Go 的版本, 根据 Go 的版本采取不同的策略. 针对 Go1.16.15 以上的 Go, 会使用 Go 现行的 ABI-Internal 从寄存器 R14 中获得 g 对象的地址. 在 Go1.16.15 及以下的 Go 中, 则会从 TLS 中的固定位置获得 g 对象的地址, 并随后获得运行时栈信息.

获得相应的栈信息后, 即可检测某地址是否在当前 Goroutine 栈中, 随后可结合规则 1 和规则 2 判断该 store 是否违反 Go 的逃逸不变式. 若该 store 违反了逃逸不变式, 就会结合该指令的地址获得其所在的函数, 并向 log 文件中输出相应的出错信息, 包括该指令的地址、所在的函数、违反不变式的原因以及当前的运行时栈信息. 这些信息可以在之后帮助开发者更快地找到问题.

3.4 采用多种措施减少误报

为了减少误报, DBI-Go 主要采取了以下措施.

(1) 措施 1: 过滤掉非 Go 函数. Go 的运行时最终以静态链接库的形式和用户代码链接成可执行文件, 其中除了 Go 函数外还包括许多汇编和 C 函数. 汇编和 C 函数不遵守 Go 的 ABI 约定, 对这些函数进行分析会得到错误的结果. 同时, 这些非 Go 函数也不遵守 Go 的逃逸不变式, 因此也无需对其进行分析. Go 的代码以包 (package) 的形式进行管理, 每个函数都有其所在的包. 基于此观察, DBI-Go 采用基于模式匹配的方法, 通过函数名判断每个函数是否在某个包内, 并据此过滤掉所有非 Go 函数.

(2) 措施 2: 过滤掉 Go 运行时函数. Go 除了会在用户代码中生成若干与运行时管理相关的代码外, 还会使用 runtime 包中的运行时函数来进行运行时管理. 这些运行时函数会产生若干违反 Go 逃逸不变式的 store 操作, 但这些操作由 Go 的运行时保证了其安全性. 因此在 DBI-Go 的实现中会过滤掉 runtime 包中的函数以避免误报.

(3) 措施 3: 过滤掉非指针 store. 利用第 3.2.3 节中的方法, DBI-Go 可以恢复 Go 二进制中 store 指针的语义, 过滤掉不包含指针的对象的 store. 使用该措施可以大大降低将非指针类型诸如 int、uintptr 等当作指针从而带来的误报, 提升 DBI-Go 的分析精度.

以上措施不仅可以降低误报, 还降低了 DBI-Go 的整体开销. 通过上述措施, 我们提升了 DBI-Go 的精度和分析效率 (详见第 4.3 节).

4 实验评估

对 DBI-Go 的实验评估在 x86-64 的机器上进行. 实验环境如下所示.

- 操作系统: Ubuntu 22.04.3 LTS (GNU/Linux 5.15.0-48-generic x86_64).
- CPU: 2 × AMD EPYC 7763 64-Core Processor.
- 内存: 1.0 TiB.
- 涉及的 Go 版本: Go1.11 至 Go1.20.5.

实验试图回答以下研究问题.

(1) DBI-Go 对已知漏洞的覆盖情况如何, 能否发现新的漏洞?

(2) DBI-Go 插桩的回调函数带来的额外开销有多少, 是否满足了轻量快速的目标?

(3) DBI-Go 的适用性如何, 能否在不同的 Go 版本上正常使用?

4.1 有效性测试

为了对 DBI-Go 的漏洞覆盖率进行测试, DBI-Go 测试了图 1 中目前所有已知的社区例子. 针对图 1 中目前可以复现且有着复现例子的 issue, 我们使用对应的 Go 版本将这些例子编译, 并之后使用 DBI-Go 插桩. 图 1 中提供

复现代码并可复现的 issue 共有 5 个, 分别为 issue#29000[37], issue#31573[38], issue#44614[15], issue#47276[39]和 issue#54247[14]. 其涉及的 Go 版本为 Go1.11 至 Go1.17, 年份跨度为 2018–2022 年. 最终的结果显示, DBI-Go 的漏洞覆盖率较高, 其可以检测出图 1 中所有可复现的例子中的违反 Go 逃逸不变式的 store 指令. 输出的 log 文件相比于 Go 原始 GC panic 时产生的信息可以更清晰地展示出产生错误的指令及其所在的函数, 可以帮助更快定位原因. 以 issue#44614[15]为例, 其简化版代码可见代码 3. 图 8(a) 为社区 issue 中 Go GC 的 log, 图 8(b) 为 DBI-Go 的 log. 相比于 GC 的 log, 其可以更精确的显示问题的产生地, 比如二进制指令地址以及所在的函数. 若能结合 DWARF 信息, 还可以找到对应的源代码的位置, 更便于问题定位.

```
runtime: marked free object in span
0x7f4ce673e548, elemsize=64
freeindex=7 (bad use of
unsafe.Pointer? try -d=checkptr)
0xc000456000 alloc marked
0xc000456040 alloc marked
0xc000456080 alloc marked
0xc0004560c0 alloc marked
0xc000456100 alloc marked
0xc000456140 alloc marked
0xc000456180 alloc marked
0xc0004561c0 alloc marked
0xc000456200 free  unmarked
0xc000456240 free  marked   zombie
…
```

(a) issue 中 GC log

```
…
store stack pointer out of current g stack
frame at 0x48b015 in main.global2stack:
store 0xc0002cbf60 from rcx to
0xc000010028;
current g stack status:
lo: 0xc0002cb000;
hi: 0xc0002cc000;
stackguard0: 0xc0002cb3a0
user program terminated with code 0
```

(b) DBI-Go log

图 8 GC log 与 DBI-Go log

Go 的标准库 (std) 和编译工具链 (cmd) 提供了大量测试用例和 Benchmark, 覆盖了其中的众多常用 API. Go 的标准库和编译工具链中共有 277 个包提供了测试用例. 我们使用最新版本的 Go (Go1.20.5), 将这些测试用例和 Benchmark 编译成可执行文件, 并随后使用 DBI-Go 进行检测. 结果表明, 在这 277 个包中的 276 个没有发现问题, 然而, 在 syscall 包中, DBI-Go 发现 Go 在处理切片的字面量时将栈上的数组地址存到了全局变量中. 它的简化版示例如图 9 所示. LEAQ 0x8(SP), AX 指令将栈对象的地址 0x8(SP) 存入了寄存器 AX 中, 接下来的 MOVQ 指令将该栈地址 store 到了某全局变量处. 该 store 违反了规则 1 并被 DBI-Go 所捕获.

```
testCases := [...][][]int{
    {{42}},
    {{1, 2}},
    {{3, 4, 5}},
    {{}},
    {{1, 2}, {3, 4, 5}, {}, {7}},
}
for _, testCase := range testCases {
    use(testCase)
}
```

经Go编译器编译

```
…
CMPL $0x0, runtime.writeBarrier(SB)
JNE 0x4585c8
LEAQ 0x8(SP), AX
MOVQ AX, 0x68e9a(IP) 此处的store违反逃逸不变式
JMP 0x4585d9
LEAQ 0x68e91(IP), DI
LEAQ 0x8(SP), AX
CALL runtime.gcWriteBarrier(SB)
…
```

图 9 syscall 包中发现的违反 Go 逃逸不变式的 store

目前该问题已经在 golang-nuts 中得到 Go 官方维护人员的确认 (https://groups.google.com/g/golang-nuts/c/YZVFzwnPixM), 并已向社区提交 issue, 有待 Go 官方的进一步修复 (https://github.com/golang/go/issues/61730).

除了上述对使用 Go 原生逃逸分析算法的编译器生成的二进制的测试, 我们还将 DBI-Go 用于评测一个在 Gollvm (一个基于 LLVM 的 Go 编译器) 上重构的 Go 逃逸分析算法. 通过用 DBI-Go 检测经重构逃逸分析算法后的 Gollvm 生成的二进制中是否有非法的内存引用, 来判断重构后的新逃逸分析算法的正确性, 并辅助开发人员进行 Debug. 在实际评测中, 用 DBI-Go 可以发现新逃逸分析算法引起的内存分配问题. 以代码 5 和代码 6 为例, 代码 5 中的 New 函数将字面量 prefixError{}的地址返回出函数; 代码 6 中, 对象 b 的地址被全局对象 gm 获取, 根据逃逸不变式 2 它们理应堆分配. DBI-Go 发现重构后的逃逸分析算法在特定场景下会将代码 5 中本应在堆中分配的 prefixError{}以及代码 6 中本应在堆中分配的 b 均分配在栈上. 通过这些例子, DBI-Go 帮助新逃逸算法的开发

者修复了重构后的逃逸算法上的 Bug.

代码 5. DBI-Go 发现的新逃逸算法出错的例子 1.

```
1. package nomain
2. type prefixError struct{s string}
3. func New(f string, x ...interface{}) error {
4.    // 误将字面量 prefixError{}分配到栈上
5.    return & prefixError{}
6. }
7. func (e *prefixError) Error() string {
8.       return “1” + e.s
9. }
```

代码 6. DBI-Go 发现的新逃逸算法出错的例子 2.

```
1. var gm map[string]interface{}
2.
3. func Test() {
4.    // 误将 b 分配到栈上
5.    var b int
6.    gm[“1”] = & b
7. }
```

4.2 额外开销

4.2.1 额外运行时开销

相比于只需在运行前执行一次的静态分析和初始化, 插桩的回调函数带来的额外运行时开销在反复执行时会占据主要比例, 这些额外的运行时开销主要包括以下几部分: (1) 调用回调函数的开销; (2) 获取 Go 的运行时信息的开销; (3) 利用规则 1 和规则 2 进行运行时验证的开销. 为了了解这些额外运行时开销的影响, 我们使用第 4.1 节中所述的 Go 标准库和编译工具链中的 277 个包中的测例进行了测试. 最终, 记录了插桩的回调函数带来的额外运行时开销相比于直接执行时所花费的开销的比值. 为了表述方便, 在下文使用 $R_{c/o}$ 表示额外的运行时开销相比于直接执行时所花费的开销的比值.

为了了解 $R_{c/o}$ 的分布, 对得到的额外开销数据使用了 KDE (kernel density estimation, 核密度估计, 一种用于估计随机变量的概率密度函数的非参数方法)[40, 41]估计了 $R_{c/o}$ 在这 277 个包中的分布密度, 如后文图 10 所示. 该曲线的波峰在 $R_{c/o}$ 约为 0.25 处达到, 且绝大多数的额外开销相比于原生开销的比值均小于 2 (93.3%) 只有在极少数 store 密集型的程序中该比值才会大于 4 (2.8%). 产生额外的 2 倍运行时开销是可以承受的.

4.2.2 额外初始化开销

在前文中提到, 由于初始化部分只需在运行前执行一次, 因此其相比于可以反复执行的运行时开销可以忽略. 但用于初始化的该部分开销在使用时也会对总时间造成影响, 因此对该部分开销的测试也是必要的. 额外的初始化开销包括以下几部分时间: (1) Pin 加载用户二进制的开销; (2) 反汇编的开销; (3) 使用静态分析, 利用 Go 的写屏障机制恢复 Go store 语义的开销. 使用和第 4.2.1 节相同的测试集和测试方式, 通过记录额外的初始化开销与原生开销的比值, 并使用 KDE 估计分布密度, 可以得到图 11. 为了表述方便, 在下文使用 $R_{i/o}$ 来表示额外的初始化开销相比于原生开销的比值.

图 11 有两波波峰, 第 1 波大约在 $R_{i/o}$ 为 4 处, 另一波对应的 $R_{i/o}$ 则超过了 100. 相比于原生开销 100 倍的额外

初始化开销是惊人的. 为了了解该部分比值为何如此之高, 我们将 $R_{i/o}$ 大于 100 的部分单独进行分析. 通过分析发现, 这部分例子原生开销很小, 均不超过 10 ms, 与其相对应的 DBI-Go 的初始化开销均在 500 ms 左右, 如图 12 所示, 仅在极个别例子上初始化开销超过了 1 500 ms. 这表明 DBI-Go 的初始化开销的下限在 500 ms 左右, 因此在遇到原生开销很小, 只有几毫秒的测例时显得 $R_{i/o}$ 很大. 但实际上, 500 ms 的初始化开销是完全可以接受的.

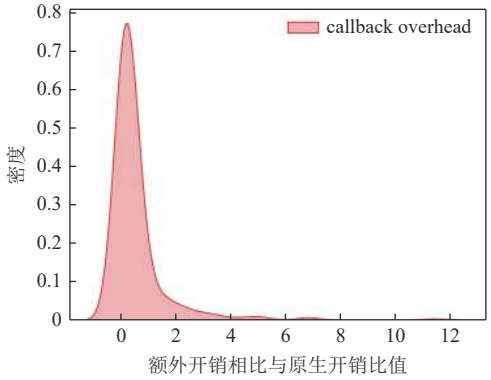


图 10 $R_{c/o}$ 的核密度分布曲线

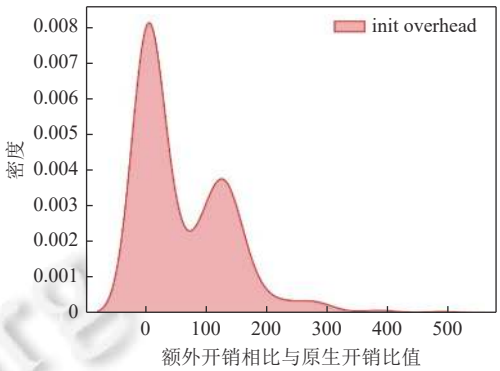


图 11 $R_{i/o}$ 的核密度分布曲线

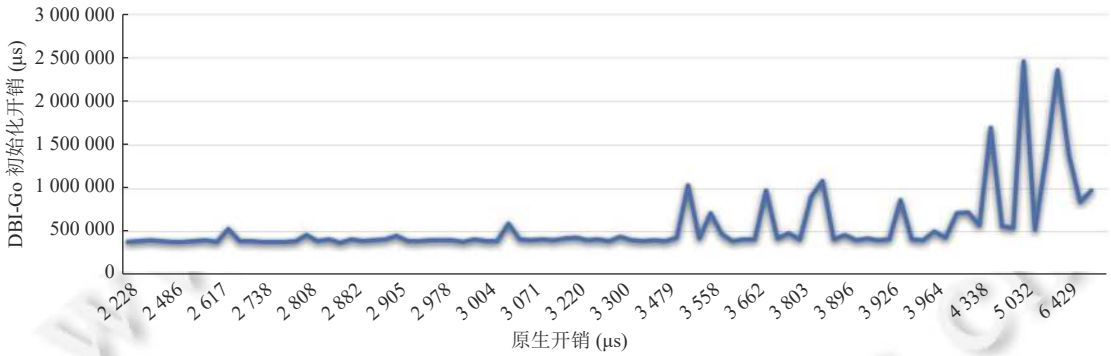


图 12 比值大于 100 的额外初始化开销与原生开销

4.3 误报率测试

为了验证 DBI-Go 所利用的 Go 写屏障机制以及第 3.4 节中的两个措施对误报率的影响, 对这些措施进行了单独或组合的测试. 在下文中, 使用“措施 1” 来代表第 3.4 节中的“过滤掉非 Go 函数” 措施; 使用“措施 2” 来代表第 3.4 节中的“过滤掉 Go 运行时函数”措施; 使用“措施 3”来代表使用第 3.4 节中的“过滤掉非指针 store” 措施; 使用“无”代表不使用任何措施, 直接插桩 Go 二进制中的所有 store. 我们使用 Go 的标准库和编译工具链提供的 277 个包, 测试方法与第 4.1 节相同.

最终的测试结果如表 1 所示. 从中看出, 单独使用措施 1 和 2 都没有效果. 这是因为目前 G 的二进制中都包含大量的运行时管理函数以及汇编函数等非 Go 函数, 单独过滤运行时函数或者非 Go 函数无法消除在单一二进制 (包) 上的误报. 从表 1 中可以看出, 同时使用措施 1 和 2 相比单独的措施 1 或 2 可以大大降低误报的包的数量, 但此时误报率仍然较高, 这是因为此时还没有恢复 Go 二进制中 store 指针的语义, 仍对所有 Go 用户代码中的 store 进行检查. 单独使用措施 3 的误报率也较高, 原因在于 Go 运行时中有诸多违反 Go 逃逸不变式的 store, 但运行时保证了其安全性. 同时使用措施 2 和措施 3 可以带来最低的误报率, 在测试的 277 个包中误报率为 0. 在测试中, 措施 1 无法在措施 3 的基础上进一步降低误报, 这是因为 Go 编译器只会对 Go 函数插入写屏障, 因此使用措施 3 就潜在的消除了所有非 Go 函数带来的影响. 虽然措施 1 无法在措施 3 的基础上进一步降低误报率, 但并不

意味着措施 1 是无用的. 措施 1 可以让措施 3 的静态分析阶段跳过诸多无意义的非 Go 函数, 降低初始化阶段所带来额外开销, 因此措施 1 也是必不可少的. 为了揭示措施 1 对减少初始化开销的作用, 我们只采用措施 2 和措施 3, 不采用措施 1 去重复第 4.2.2 节中的测试, 结果显示在这 277 个包中, 总初始化时间增加了约 20.14%.

表 1 各个措施误报率测试结果

减少误报的措施	总包数	报错的包数	误报的包数	误报率 (%)
无	277	277	276	99.64
措施1	277	277	276	99.64
措施2	277	277	276	99.64
措施1+措施2	277	24	23	95.83
措施3	277	274	273	99.63
措施3+措施1	277	274	273	99.63
措施3+措施2	277	1	0	0.00
措施3+措施1+措施2	277	1	0	0.00

4.4 开源项目测试

为了了解真实世界 Go 项目中是否有违反 Go 逃逸不变式引发的内存安全问题, 我们选取了 18 个在各个领域具有代表性的 Go 开源项目来进行测试, 这些开源项目涉及 Web、数据库、分布式系统、边缘计算、云存储等多个领域. 选取的仓库如表 2 所示. 测试时使用和第 4.1 节相似的测试方法: 以包为单位, 将这些项目提供的测试用例和 Benchmark 编译成可执行文件, 并随后使用 DBI-Go 进行检测. 在这 18 个开源仓库中共有 1 730 个包提供了测试用例或者 Benchmark. 最后的测试结果显示, DBI-Go 在这些仓库中并未检测到问题.

表 2 测试的代表性 Go 开源项目

类型	仓库名	简介	可测试的包数
Web	hugo	轻量 Web 框架	123
	beego	Go 语言 Web 框架	54
	go-restful	用 Go 构建的 REST 风格的 Web 服务包	1
	websocket	WebSocket 协议的 Go 实现	1
代理	v2ray-core	网络代理工具	64
数据库	dgraph	分布式的、可扩展的图数据库管理系统	84
	etcd	分布式键值数据库	93
git服务	cli	GitHub 官方命令行工具	170
	gogs	自托管 Git 服务	24
边缘计算	kubeedge	Kubernetes 原生边缘计算框架	66
存储管理	open-local	云原生本地存储管理系统	9
	rclone	云存储管理	133
依赖注入框架	IOC-golang	依赖注入框架, 便于搭建任何 Go 应用	55
序列化	protobuf-go	Google 数据交换格式	43
	json-iterator/go	encoding/json 包的高性能替代	10
分布式	grpc-go	gRPC 的 Go 语言实现	108
容器管理	kubernetes	生产级容器调度和管理	680
日志管理	zap	快速、结构化、分级的日志	12
		合计	1 730

4.5 适用性测试

上述测试中针对额外运行时开销、Go 标准库以及编译工具链的测试中所使用的 Go 编译器为当前的最新版

本 (Go1.20.5). 在对漏洞覆盖率测试中, 我们会根据图 1 中每个 issue 所描述的 Go 版本来选取相应版本的 Go 编译器. 最终结果显示 DBI-Go 在这些 Go 版本中均可正常运行 (Go1.11 至 Go1.20.5).

5 DBI-Go 的局限性讨论

综合来看, DBI-Go 仍有进一步优化的优化空间. 对此, 本节列出来了目前 DBI-Go 的一些局限性及一些可能的解决思路.

• 动态分析工具的代码覆盖率问题. DBI-Go 是基于二进制插桩的动态检测器, 因此它无法检测 Go 程序中未被执行的路径中的 store 指令. 这是所有动态分析工具不得不面对的问题. 为了改善这种情况, 可以使用基于代码覆盖率的模糊测试等技术来提升代码覆盖率.

• 现有规则可进一步扩充. DBI-Go 的运行时验证部分基于规则 1 和规则 2. 其中规则 2 基于 Go 的逃逸不变式 2 “指向栈对象的指针生命期不可超出该栈对象”总结得到. 目前规则 2 所考虑的“生命期超出该栈对象”的情况分为 5 种: 全局对象、堆对象、更深的栈对象、栈帧更深的栈对象以及其他 Goroutine 栈对象. 但实际上“生命期超出该栈对象”还有其他情况, 比如更浅的作用域中的栈对象等. 由于这些情况在二进制中不好识别, 目前规则 2 未考虑. 因此规则 2 目前仍有进一步扩充的空间.

• 误报无法完全消除. 目前基于 gcWriteBarrier 的 store 语义恢复机制只适用于栈到堆或全局的 store 指令. 违反逃逸不变式的栈到栈的 store 指令的检测目前仍需要插桩二进制中的所有 store 指令. 尽管目前已做了一些筛选, 比如不插桩运行时函数中的 store, 跳过汇编函数等. 但由于缺乏 Go 的高层语义信息, 比如类型描述符, 此时仍有可能带来误报. 同时, gcWriteBarrier 是 Go 编译器在编译期间在源代码层级上增加的调用, 其以对象为粒度, 而非二进制中的指令. 若某对象中既包含指针类型也包含非指针类型 (如 uintptr), 当该非指针类型的值等于某个栈指针时, 由于 DBI-Go 假定其为指针, 此时仍有可能产生误报. 若要杜绝此类问题, 一种可能的方法是利用 Go 运行时中的 bitmap. 该 bitmap 可以指示内存中何处是指针, 何处不是指针, Go 的 GC 会利用该 bitmap 进行标记和清扫. 但引入 bitmap 会导致 DBI-Go 和 Go 的不同版本运行时强绑定, 降低其可扩展性, 同时还会进一步增大 DBI-Go 的额外开销. 因此, 综合考虑, 由于目前 DBI-Go 的误报率在可接受的范围中, 故没有引入 bitmap 机制.

• ABI 需要根据不同 Go 版本进行适配. DBI-Gos 实现的其中关键一点在于利用 Go 的 ABI 获得 Goroutine 运行时栈信息. 目前 Go1.17 及以上版本采用现行的 ABI-Internal, 而 Go1.16.15 及以下则是另一套 ABI. 为了保证适用性, 目前 DBI-Go 针对不同版本做了适配. 若未来 Go 的 ABI 发生进一步变化, 则 DBI-Go 也需要进行相应的适配. 不过需要注意的是, Go 运行时中 Goroutine 的栈结构比较稳定, 其结构从 2014 年的 Go1.4 到 2023 年最新的 Go1.20 版本均未发生变化. 这意味着未来只需对 ABI 进行适配, 而使用 Goroutine 栈信息的运行时验证部分则无需更改.

• 依赖 Go 的写屏障机制的正确性. DBI-Go 的分析目前依赖于 Go 编译器在编译期间插入的 gcWriteBarrier 函数, 若 Go 编译器由于误判等原因没有对某 store 指针到堆的操作插入 gcWriteBarrier 函数, 则 DBI-Go 无法判断是否有违反逃逸不变式的情况.

• 需要二进制上的符号信息. DBI-Go 需要二进制中的符号信息才能正常工作, 包括函数名、全局变量名等. 目前 DBI-Go 无法在完全剥离符号信息的二进制上进行分析.

• 额外开销可进一步降低. 在额外开销方面, 目前回调函数的额外开销仍有继续优化的空间. Pin 作为一个通用的动态二进制分析框架, 没有针对 Go 的特定优化. DBI-Go 在设计实现时本着快速原型化的理念, 也未针对 Go 做大量优化. 目前 DBI-Go 所做的优化有使用基于匹配的方式跳过 Go 的运行时函数和一些非 Go 函数 (如一些汇编函数、C 函数) 以及基于 gcWriteBarrier 机制只插桩可能 store 指针的指令. 未来基于 Go 语言的特性可以探索更多的优化方式, 减少 Pin 插桩带来的额外开销.

• 无法自动修复漏洞. DBI-Go 只是一个 Go 二进制中的漏洞检测器, 它并不能自动修复检测到的漏洞. 漏洞产生的原因有很多种, 比如逃逸分析的错误、编译优化的错误等. 通过 DBI-Go 检测出的漏洞需要 Go 编译器开发人员的进一步分析和修复. 但 DBI-Go 的 log 可以帮助开发人员更快的确认漏洞出现的位置及原因.

• 有待进一步大规模测试. 目前 DBI-Go 在真实世界开源项目上的测试尚未发现问题, 只在 18 个仓库上进行了测试, 覆盖面不足. 未来期望能够进行更大规模的测试, 以期找出开源仓库中的问题, 帮助改善 Go 语言软件的内存安全性, 提升其可靠性.

6 相关工作

6.1 动态二进制分析

动态二进制分析框架, 如 Pin[21]、Valgrind[42]、DynamoRIO[43], 可以用于插桩任意指令来执行动态分析. 这些工具为研究者们的分析带来了很大的便利. Amitabha 等人[44]研究了如何有效地插桩 x86 机器码中的内存访问以支持软件事务内存和分析. Patil 等人[45]基于 Pin 设计了 Pinplay, 一个基于执行捕获和确定性重放的并行程序分析框架. Zhong 等人[17]基于 DynamoRIO 设计了一个动态二进制工具, 用于检测 Go 中的并发问题.

基于动态二进制分析的漏洞检测包括前面提到的并发漏洞分析、污点分析[46]、逆向工程[47]和执行重放[48]等.

6.2 内存漏洞检测

目前已有的内存相关的漏洞检测工作主要是针对 C/C++这种具有弱静态类型系统的编程语言来进行的, 因为强制类型转换、任意指针的存在使得悬空引用、边界溢出等内存漏洞更容易发生, Song 等人于 2013 年[30]通过系统性地建立内存损坏的一般模型, 揭示了 C/C++容易遭受内存漏洞的主要原因. 现有的内存漏洞检测工作依赖于静态程序分析或动态程序分析来进行.

静态检测: 分析程序 (源) 代码, 并生成对于所有可能的代码执行都是保守正确的结果. 静态检测的一部分工作基于形式化验证: Clarke 等人提出了一种使用有界模型检查 (BMC) 对 ANSI-C 程序进行形式化验证的工具[49]来检测内存问题. 更多的静态检测工作则是基于符号执行: CUTE[50]通过结合符号执行和具体执行, 将内存图作为输入来执行自动化测试; EXE[51]是利用符号执行来自动生成导致实际代码崩溃的输入, 进行快速的错误定位; Klee[52]则是通过符号执行来自动生成测试.

动态检测: 通过分析单个程序的执行, 并输出仅对单个运行有效的精确分析结果. 消毒器 (sanitizer)——静态插入运行时监视器, 并在运行时进行检测——是动态检测工具的典型代表. Serebryany 等人在 2012 年提出了 AddressSanitizer (ASAN)[28], 它通过插桩应用程序中的内存访问操作, 在运行时建模影子内存, 从而能识别缓冲区溢出、悬垂指针、内存泄漏等内存漏洞. 由于它能够在不牺牲完备性的情况下实现了检测效率, AddressSanitizer 已经被集成到许多常用的编译工具链中, 包括针对 C/C++的编译器 GCC、LLVM, 以及 Go 语言编译器. 但 ASAN 目前在 Go 编译器中的使用场景受限, 仅能检测 Go 语言中和 C 语言进行交互的相关代码上的内存错误[53], 对于纯 Go 语言代码尚不支持. ASAN 与 DBI-Go 的相同点在于二者都是基于插桩, ASAN 是在编译时插桩, DBI-Go 是基于动态二进制插桩. 区别在于 ASAN 的设计目的是检测诸如缓冲区溢出之类的通用的内存漏洞, 不能检测 Go 中违反 Go 逃逸不变式的 store; 且其所能检测的内存漏洞只有在被触发时才能发现 (如内存被释放、指针被解引用时), 此时 Go 程序可能已经崩溃. 而 DBI-Go 则是针对 Go 语言专门设计, 利用 Go 的逃逸不变式这一独特特性, 可以在内存隐患发生的第一现场就报错 (比如将栈地址存入堆时). 因此即使 ASAN 可以支持纯 Go 的代码, 其也无法代替 DBI-Go. 类似地, 还有许多使用类似方式进行针对其他问题检测的漏洞检测工具, 如用于检测未初始化内存的使用情况的 MemorySanitizer[54], 用于检测数据竞争的 ThreadSanitizer[55], 利用动态内存检查来检测对象有效性的 EffectiveSan[56] 等. Song 等人[29] 则是在 2019 年对 Sanitizing 这种技术进行了对比总结, 描述了不同的 Sanitizer 工具的性能和可扩展性. 此外, 还有部分工作通过修改运行时系统来实现运行时检测, 如 DieHard[57]、SoftBound[58].

6.3 Go 的漏洞检测

目前针对 Go 的漏洞检测已有许多工作. Lange 等人[19]为 Go 中的消息传递机制进行建模, 为 Go 的消息传递机制提出了一个验证框架. Lauinger 等人[33]提出了 go-safer, 一种全新的静态分析工具, 用来识别 Go 源代码中对

unsafe 包的不安全使用. Wang 等人[59]设计了 HERO, 用于检测 Go 中依赖管理导致的问题. Liu 等人[18]提出了 GCatch, 用于自动检测和修复 Go 中的并发问题. Chabbi 等人[31] 使用现有的数据竞争检测器在 Uber 项目中发现了超过 2000 个数据竞争. Li 等人[60]设计了 CryptoGo, 用于检测 Go 中和加解密相关 API 的误用. Zhong 等人[17]首次提出了使用动态二进制插桩的方式检测 Go 中的并发问题. 目前与 Go 漏洞检测相关的工作多数集中在对用户代码的漏洞的检测, 且多与并发有关. 本文提出的 DBI-Go 是首个验证 Go 编译器生成的代码的是否满足 Go 逃逸不变式的工具.

6.4 Go 的逃逸分析

目前 Go 中和逃逸分析的相关工作较少. Google 曾在 2015 年总结了当时 Go 逃逸分析的缺陷, 指出了其分析的保守之处[61]. Wang 等人[20]则注意到了 Go 逃逸分析的一些保守之处, 其工作使得一些对象可以绕过 Go 的逃逸分析, 从而节省堆内存的使用.

7 总结与展望

本文主要提出了 DBI-Go, 一个用于 Go 应用程序的新型漏洞检测工具. DBI-Go 使用静态分析辅助动态二进制插桩的分析方法, 以 Go 二进制文件为输入, 检测 Go 编译器生成的代码中是否有违反 Go 逃逸不变式的 store. DBI-Go 使用静态分析的方法, 结合 Go 的 gcWriteBarrier 机制恢复 Go 的 store 语义. DBI-Go 的运行时回调函数在运行时结合 Go 的 ABI 约定获得 Go 的运行时栈信息来辅助分析. DBI-Go 使用约 1 000 行 C++代码实现, 为比较轻量的检测工具.

实验表明, DBI-Go 可以检测出目前 Go 社区已经确认的问题, 呈现了较高的漏洞覆盖率, 同时 DBI-Go 还成功检测出一个之前未知的问题, 目前该问题已经得到 Go 官方的确认并在等待进一步修复. 在实际项目上的应用还表明 DBI-Go 可以辅助开发人员对内存优化相关算法, 如逃逸分析算法, 进行优化和重构, 验证算法的正确性. 对误报率的测试则表明 DBI-Go 所采取的措施可以有效地降低误报. 实验结果还表明, DBI-Go 在不同版本的 Go 编译器编译出的二进制上都能正常工作 (Go1.11 至 Go1.20.5), 体现了较高的可扩展性. 额外开销的测试结果则表明 DBI-Go 会产生在可接受范围内的约常数倍的开销.

本文还分析了 DBI-Go 的不足之处. 未来将继续改进 DBI-Go, 以期实现更高的代码覆盖率、更高的精度以及更小的额外开销, 并将进行更大规模, 更大范围的测试, 以期找到更多漏洞, 帮助改善 Go 语言软件的可靠性和安全性.

References:

[1] Go. The Go programming language. 2023. https://go.dev/

[2] TIOBE. Programming language hall of fame. 2023. https://www.tiobe.com/tiobe-index

[3] Taylor N. 2022 Hiring report: Golang developers. 2022. https://sgp.technology/2022-hiring-report-golang-developers/

[4] Salgado PG. Garbage collector design. 2023. https://devguide.python.org/internals/garbage-collector/

[5] Schatzl T. Java garbage collection: The 10-release evolution from JDK 8 to JDK 18. 2022. https://blogs.oracle.com/javamagazine/post/java-garbage-collectors-evolution?source=:em:nw:mt::::RC_WWMK200429P00043C0061:NSL400242337

[6] Ghemawat S, Menage P. TCMalloc: Thread-caching Malloc, 2009. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[7] Fua P, Lis K. Comparing Python, Go, and C++ on the N-queens problem. arXiv:2001.02491, 2020.

[8] Blanchet B. Escape analysis for object-oriented languages: Application to Java. ACM SIGPLAN Notices, 1999, 34(10): 20–34. [doi: 10.1145/320385.320387]

[9] Whaley J, Rinard M. Compositional pointer and escape analysis for Java programs. In: Proc. of the 14th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. Denver: ACM, 1999. 187–206. [doi: 10.1145/320384.320400]

[10] Choi JD, Gupta M, Serrano MJ, Sreedhar VC, Midkiff SP. Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. on Programming Languages and Systems, 2003, 25(6): 876–910. [doi: 10.1145/945885.945892]

[11] Kotzmann T, Mössenböck H. Escape analysis in the context of dynamic compilation and deoptimization. In: Proc. of the 1st ACM/USENIX Int'l Conf. on Virtual Execution Environments. Chicago: ACM, 2005. 111–120. [doi: 10.1145/1064979.1064996]

[12] Kotzmann T, Wimmer C, Mössenböck H, Rodriguez T, Russell K, Cox D. Design of the Java HotSpot™ client compiler for Java 6. ACM

Trans. on Architecture and Code Optimization, 2008, 5(1): 7. [doi: 10.1145/1369396.1370017]

[13] Stadler L, Würthinger T, Mössenböck H. Partial escape analysis and scalar replacement for Java. In: Proc. of the 2014 Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. Orlando: ACM, 2014. 165–174. [doi: 10.1145/2581122.2544157]

[14] Go. Go/issues/54247. 2022. https://github.com/golang/go/issues/54247

[15] Go. Go/issues/44614. 2021. https://github.com/golang/go/issues/44614

[16] Go. Bad pointer. 2023. https://github.com/golang/go/blob/2fcfdb96860855be0c88e10e3fd5bb858420cfe2/src/runtime/mbitmap.go#L321

[17] Zhong CX, Zhao QD, Liu X. BinGo: Pinpointing concurrency bugs in Go via binary analysis. arXiv:2201.06753, 2022.

[18] Liu ZH, Zhu SF, Qin BQ, Chen H, Song LH. Automatically detecting and fixing concurrency bugs in Go software systems. In: Proc. of the 26th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM, 2021. 616–629. [doi: 10.1145/3445814.3446756]

[19] Lange J, Ng N, Toninho B, Yoshida N. A static verification framework for message passing in go using behavioural types. In: Proc. of the 40th IEEE/ACM Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 1137–1148. [doi: 10.1145/3180155.3180157]

[20] Wang C, Zhang MR, Jiang Y, Zhang HF, Xing ZC, Gu M. Escape from escape analysis of Golang. In: Proc. of the 42nd IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Seoul: IEEE, 2020. 142–151.

[21] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. ACM Sigplan Notices, 2005, 40(6): 190–200. [doi: 10.1145/1064978.1065034]

[22] Go. Goroutines. 2023. https://go.dev/doc/effective_go#goroutines

[23] Go. A guide to the Go garbage collector. 2023. https://go.dev/doc/gc-guide

[24] Go. Frequently asked questions (FAQ). 2023. https://go.dev/doc/faq#stack_or_heap

[25] Go. Go escape analysis invariant. 2022. https://github.com/golang/go/blob/master/src/cmd/compile/internal/escape/escape.go#L22

[26] Go. Go/test/escape.go. 2022. https://github.com/golang/go/blob/master/test/escape.go#L10

[27] Go. Go/test/escape_level.go. 2022. https://github.com/golang/go/blob/master/test/escape_level.go#L14

[28] Serebryany K, Bruening D, Potapenko A, Vyukov D. AddressSanitizer: A fast address sanity checker. In: Proc. of the 2012 USENIX Annual Technical Conf. Boston: USENIX Association, 2012. 309–318.

[29] Song D, Lettner J, Rajasekaran P, Na Y, Volckaert S, Larsen P, Franz M. SoK: Sanitizing for security. In: Proc. of the 2019 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2019. 1275–1295. [doi: 10.1109/SP.2019.00010]

[30] Szekeres L, Payer M, Wei T, Song D. SoK: Eternal war in memory. In: Proc. of the 2013 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2013. 48–62. [doi: 10.1109/SP.2013.13]

[31] Chabbi M, Ramanathan MK. A study of real-world data races in Golang. In: Proc. of the 43rd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. San Diego: ACM, 2022. 474–489. [doi: 10.1145/3519939.3523720]

[32] Go. Go internal ABI specification. 2022. https://github.com/golang/go/blob/master/src/cmd/compile/abi-internal.md

[33] Lauinger J, Baumgärtner L, Wickert AK, Mezini M. Uncovering the hidden dangers: Finding unsafe Go code in the wild. In: Proc. of the 19th IEEE Int'l Conf. on Trust, Security and Privacy in Computing and Communications. Guangzhou: IEEE, 2020. 410–417. [doi: 10.1109/TrustCom50675.2020.00063]

[34] Uber-Go. Goleak. 2023. https://github.com/uber-go/goleak

[35] Zeng B. Static analysis on binary code. 2012. https://engineering.lehigh.edu/sites/engineering.lehigh.edu/files/_DEPARTMENTS/cse/research/tech-reports/2012/lu-cse-12-001.pdf

[36] Go. The Go programming language specification. 2023. https://go.dev/ref/spec

[37] Go. Go/issues/29000. 2018. https://github.com/golang/go/issues/29000

[38] Go. Go/issues/31573. 2019. https://github.com/golang/go/issues/31573

[39] Go. Go/issues/47276. 2021. https://github.com/golang/go/issues/47276

[40] Rosenblatt M. Remarks on some nonparametric estimates of a density function. The Annals of Mathematical Statistics, 1956, 27(3): 832–837. [doi: 10.1214/aoms/1177728190]

[41] Parzen E. On estimation of a probability density function and mode. The Annals of Mathematical Statistics, 1962, 33(3): 1065–1076. [doi: 10.1214/aoms/1177704472]

[42] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, 42(6): 89–100. [doi: 10.1145/1273442.1250746]

[43] DynamoRIO. 2002. https://dynamorio.org/

[44] Roy A, Hand S, Harris T. Hybrid binary rewriting for memory access instrumentation. In: Proc. of the 7th ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments. Newport Beach: ACM, 2011. 227–238. [doi: 10.1145/1952682.1952711]

[45] Patil H, Pereira C, Stallcup M, Lueck G, Cownie J. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In: Proc. of the 8th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. Toronto: ACM, 2010. 2–11. [doi: 10.1145/1772954.1772958]

[46] Brumley D, Newsome J, Song D, Wang H, Jha S. Towards automatic generation of vulnerability-based signatures. In: Proc. of the 2006 IEEE Symp. on Security and Privacy. Berkeley: IEEE, 2006. 15–16. [doi: 10.1109/SP.2006.41]

[47] Lin ZQ, Jiang XX, Xu DY, Zhang XY. Automatic protocol format reverse engineering through context-aware monitored execution. In: Proc. of the 15th Symp. on Network and Distributed System Security. San Diego: NDSS, 2008. 1–15.

[48] Narayanasamy S, Pokam G, Calder B. Bugnet: Continuously recording program execution for deterministic replay debugging. In: Proc. of the 32nd Int'l Symp. on Computer Architecture. Madison: IEEE, 2005. 284–295. [doi: 10.1109/ISCA.2005.16]

[49] Clarke E, Kroening D, Lerda F. A tool for checking ANSI-C programs. In: Proc. of the 10th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Barcelona: Springer, 2004. 168–176. [doi: 10.1007/978-3-540-24730-2_15]

[50] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes, 2005, 30(5): 263–272. [doi: 10.1145/1095430.1081750]

[51] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: Automatically generating inputs of death. ACM Trans. on Information and System Security, 2008, 12(2): 10. [doi: 10.1145/1455518.1455522]

[52] Cadar C, Dunbar D, Engler DR. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224. [doi: 10.5555/1855741.1855756]

[53] Go. Go 1.18 release notes. 2022. https://tip.golang.org/doc/go1.18#go-command

[54] Stepanov E, Serebryany K. MemorySanitizer: Fast detector of uninitialized memory use in C++. In: Proc. of the 2015 IEEE/ACM Int'l Symp. on Code Generation and Optimization. San Francisco: IEEE, 2015. 46–55. [doi: 10.1109/CGO.2015.7054186]

[55] Serebryany K, Iskhodzhanov T. ThreadSanitizer: Data race detection in practice. In: Proc. of the 2009 Workshop on Binary Instrumentation and Applications. New York: ACM, 2009. 62–71. [doi: 10.1145/1791194.1791203]

[56] Duck GJ, Yap RHC. EffectiveSan: Type and memory error detection using dynamically typed C/C++. ACM SIGPLAN Notices, 2018, 53(4): 181–195. [doi: 10.1145/3296979.3192388]

[57] Berger ED, Zorn BG. DieHard: Probabilistic memory safety for unsafe languages. ACM SIGPLAN Notices, 2006, 41(6): 158–168. [doi: 10.1145/1133255.1134000]

[58] Nagarakatte S, Zhao JZ, Martin MMK, Zdancewic S. SoftBound: Highly compatible and complete spatial memory safety for C. ACM SIGPLAN Notices, 2009, 44(6): 245–258. [doi: 10.1145/1543135.1542504]

[59] Wang Y, Qiao L, Xu C, Liu YP, Cheung SC, Meng N, Yu H, Zhu ZL. Hero: On the chaos when PATH meets modules. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering. Madrid: IEEE, 2021. 99–111. [doi: 10.1109/ICSE43902.2021.00022]

[60] Li WQ, Jia SJ, Liu LM, Zheng FY, Ma Y, Lin JQ. CryptoGo: Automatic detection of Go cryptographic API misuses. In: Proc. of the 38th Annual Computer Security Applications Conf. Austin: ACM, 2022. 318–331. [doi: 10.1145/3564625.3567989]

[61] Vyukov D. Go escape analysis flaws. 2015. https://docs.google.com/document/d/1CxgUBPlx9iJzkz9JWkb6tIpTe5q32QDmz8l0BouG0Cw

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

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

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, 智能计算, 数据计算, 量子计算.

丁伯尧(1999－), 男, 博士生, CCF 学生会员, 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.