Detecting Go Concurrency Bugs: Which Tool Performs Best?

Yantong Li

Master of Science
School of Informatics
University of Edinburgh
2023

Abstract

Concurrency bugs present significant challenges in Go programming, usually arising from the incorrect utilization of concurrency mechanisms. These bugs can reduce program reliability and performance, undermining the application and development of the Go language. This paper conducts a comprehensive evaluation of four popular Go bug detection tools: GFuzz, Goleak, GCatch, and GoAT. These tools are assessed using the GoBench benchmark suite, which encompasses go concurrency bugs from various categories. The experiment is conducted based on evaluation metrics such as tool deployment difficulty, bug detection capability, and execution load. Through analysis on the results, the strengths and weaknesses of these tools are identified, and their recommended use cases are demonstrated. Our study aims to provide valuable insights for Go developers in selecting testing tools to enhance program reliability. Furthermore, by evaluating existing tools, we make contribution to the design of more advanced bug detection tools in the future.

Research Ethics Approval

Declaration

(*Yantong Li*)

Acknowledgements

I would like to express my sincere gratitude to Professor Rajan, my supervisor, for her continuous guidance throughout the project. Her valuable feedback and insightful advice during our meetings have significantly contributed to the completion of my work. I also extend my thanks to the lecturers and staff who have played a vital role in my pursuit of the master's degree. Your dedicated teaching and assistance have equipped me with a wealth of knowledge and skills. Additionally, I am grateful for the strong support and companionship provided by my girlfriend and friends. Your presence in my life has added much more vibrancy to my life experience.

Table of Contents

Chapter 1

Introduction

1.1 Background

Go[12], also known as Golang, is a programming language developed by Google. In recent years, Go has gained large popularity due to its efficiency, high-performance and simplicity[10]. It has fostered a rapidly growing developer community, positioning Go in fast development[8]. Known for its suitability in cloud services development, such as containerization tool Docker[6] and orchestration system Kubernetes[20], Go has shown its strengths in software development and holds a promising future[22].

At the core of Go's high-performance and high concurrency lies the design of goroutines[10], which is fundamental to Go's intended usage. Goroutines leverage explicit message passing through channels for communication, offering a simpler and more reliable alternative to traditional shared memory mechanisms[5]. Furthermore, Go also supports traditional shared memory primitives, offering developers diverse options.

However, like many other concurrent programming languages[23][28], the challenges posed by concurrency bugs are unavoidable for Go. With a variety of concurrency primitives supported, Go introduces more complexity to the bug detection[31][33]. Beyond the classic issues coming from improper use of resource locks, the Go-Specific primitives, such as channels and anonymous functions, also contribute to the emergence of bugs. These bugs can lead to unexpected program blocking or produce outcomes that deviate from expectations[31]. The presence of such bugs obviously damages both the reliability and performance of Go programs, retarding the growth of Go language. Hence, effective bug detection tools become important to ensuring the expected execution of Go programs, as it can expose hidden bugs during test phase, avoiding potential losses after release.

1.2 Motivation

While a large portion of existing bug detection tools target traditional programming languages like C/C++ and Java[19][18][26], focusing on shared memory accessing related bugs, the options of bug detection tools designed specifically for Go is limited[25][3]. Given the importance of effective bug detection methods in ensuring the reliability of concurrent programs, it becomes necessary to understand the strengths and weaknesses of existing bug detection tools to select the most suitable ones for program analysis.

Interestingly, there is a shortage of comprehensive cross-tool comparisons in the existing literature. Tool developers often present their products in scenarios that highlight their tool's strengths, creating a somewhat fragmented understanding of their capabilities. Consequently, a comprehensive evaluation of these tools, clarifying their features and use cases, becomes both meaningful and crucial. Furthermore, the study of existing bug detection tools offers insights that can be valuable for the tools' refinement, contributing to the reliability and robustness of Go programs and the development of Go language.

1.3 Main Contribution

This paper conducts a comprehensive study of existing Go concurrency bug detection tools. In doing so, we achieve the following key objectives:

1. **Evaluation of Bug Detection Tools:** Our work focus on an extensive evaluation of four popular Go bug detection tools. To ensure robust evaluation, we employ these tools to perform bug detection tasks on the widely used GoBench[33] dataset. This dataset provides various samples that enable comprehensive performance assessment. Our evaluation focuses on multiple aspects, including:
    - Installation and Execution Difficulty.
    - Bug Detection Ability.
    - Execution Time and Overhead.
2. **Comprehensive Analysis:** We extend our study to offer a detailed comparison of the detection results produced by the four tools. We conduct in-depth evaluation on their performance across different types of bugs and analyze the effectiveness of the tools' detection algorithms based on the code. By highlighting the strengths

and limitations, we provide an all-sided understanding of detection capability and recommended use cases for each tool. Additionally, based on our findings, we offer insightful suggestions for the refinement of future Go concurrency bug detection tools.

Overall, this paper's main contribution lies in the comprehensive evaluation of existing Go concurrency bug detection tools. Our work serves to enable developers to make informed choices when selecting tools to enhance the reliability and quality of their concurrent programs and advance the understanding of the state-of-the-art techniques in Go bug detection.

Chapter 2

Background and Related Work

2.1 Go Concurrency Programming

Concurrency programming is a key application in Go that incorporates shared memory accessing found in traditional languages like Java and C[28][17] and message passing techniques seen in Erlang[29]. The innovation of Go's concurrency programming lies in the utilization of goroutines[10], which are lightweight threads managed by the runtime scheduler. In comparison to traditional threads, the creation cost of goroutines is small, allowing hundreds of goroutines to run on a single machine. Within Go, the main method of communication between goroutines is through channels[21]. Channels can be created and closed by goroutines and can also have buffer sizes set. When buffer is empty, an attempt to receive data from the channel will block the receiving goroutine until another goroutine closes or sends data to it. Similarly, sending data to a channel with full buffer will cause the sending goroutine to be blocked. Apart from the basic communication between goroutines, Go also employs the *select-case* statements, allowing a goroutine to wait on multiple operations related to channels. A *select* can have multiple cases and an optional default branch. When more than one cases are matched, a random one will be selected to run.

Moreover, Golang supports shared memory mechanisms, which are common in traditional programming languages[25]. Go permits multiple goroutines to access the same memory and provides primitives for the protection of shared memory accessing[15], including *Mutex*(Lock/Unlock), *RWMutex*(read/write locks), *Cond*(condition variables). Additionally, the primitive *WaitGroup*, used to synchronize the completion of tasks among multiple concurrently running goroutines, is also widely used in Go. Like *pthread_join* in the C language[23], developers can utilize *WaitGroup* to coordinate the execution of goroutines, blocking the main goroutine until all goroutines in the group have completed their tasks. By employing the *Add()* method, goroutines are added to the *WaitGroup*. Goroutines within the WaitGroup use the *Done()* method to notify task completion, and the main goroutine employs the *Wait()* method to await notifications for all goroutines in the *WaitGroup*.

2.2 Go Concurrency Bugs

In Go programming, concurrency bugs frequently emerge due to the improper use of concurrency mechanisms[5]. These bugs can be categorized into two types based on their impact: blocking bugs and nonblocking bugs. Blocking bugs in the Go language encompass not only global deadlocks but also partial deadlocks where the main process finishes successfully while some goroutines are blocked[21]. Nonblocking bugs may lead to unexpected outcomes due to unanticipated behavior[33].

Misuse of resource locks is one of the causes of blocking bugs. For example, when two goroutines are waiting for each other to release the resource they hold, it results in a deadlock. This type of bug is not unique to Go and can be found in many programming languages [19][2]. Communication deadlocks is another cause of this type of bugs, which arise from the improper use of the channel mechanism[31]. For instance, if a message is sent to a channel without a buffer, but there's no other goroutine to receive it, a deadlock occurs. Additionally, the uncertainty introduced by the select-case statement contribute to the communication deadlocks, which could lead to unexpected execution path[24]. Since channel mechanism is Go-Specific, this type of bug is unique to the Go programming language[31]. Furthermore, due to Go's various concurrency primitives, there are blocking bugs produced by mixed deadlocks[33]. For instance, deadlocks can occur when some goroutines are blocked on resource locks while others are blocked on messages.

For nonblocking bugs, data races are a traditional and important cause[31][18]. For example, when multiple goroutines simultaneously operate on the same variable, different orders of these operations can lead to different outcomes. Channels can also introduce this type of bug, like receiving information from a *nil* channel, resulting in unpredictable data[10]. Additionally, bugs can arise from the misuse of anonymous functions or *WaitGroup*. For instance, invoking the *Wait()* method before *Add()* can lead to a failure in synchronization between goroutines when *WaitGroup* is used.

2.3 Go Bug Detection Tools

Bug detection has always been an important topic within the programming domain, as effective bug detection can greatly enhance stability and reliability for programs. Various approaches exist for bug detection, such as stress testing[30], which involves extensive executions to test as many feasible interleavings as possible. Alternatively, static analysis of execution paths through code can be employed[27] to assess the risk of bug existence. As an emerging programming language, Go's bug detection tools are relatively few[24]. We choose to evaluate the following four popular bug detection tools:

2.3.1 Goleak

Goleak[32] is a detection tool that focuses on the state of goroutines. For each goroutine, Goleak records a stack, which includes its state, creation function, and a full execution trace. As the program executes, Goleak gathers information about each goroutine. If the main goroutine doesn't finish its work within a predefined time threshold, Goleak reports a global deadlock. Conversely, if the main goroutine does complete, Goleak proceeds to check whether all other goroutines have also terminated successfully. If any blocked goroutines are detected, Goleak reports a partial deadlock and pinpoints the bug's cause based on the stack trace. This approach is straightforward, as it analyzes real-time program execution information to identify the presence of blocking bugs. Additionally, Goleak examines the full trace to analyze how Goroutines access variables and channels, enabling the detection of nonblocking bugs. To determine the existence of data races, Goleak cross-checks the full stack traces recorded for goroutines with runtime[14] data. It checks whether two goroutines are concurrently accessing the same variable, with at least one involving a write operation, leading to data race condition. In terms of channels, Goleak employs the trace to locate goroutines that send messages to uninitialized channels, which can lead to channel misuse bugs.

2.3.2 GFuzz

GFuzz[24] employs message reordering techniques to detect channel related concurrency bugs in Go. It targets the Go-Specific *select-case* statement, where the processing order of each case is intentionally designed non-deterministic[10]. When dealing with a substantial number of potential processing orders, unexpected execution orders might occur, leading to channel related bugs. GFuzz's solution is to deliberately alter the order of concurrent cases, guiding the test program into different execution states and thereby increasing the likelihood of triggering blocking bugs.

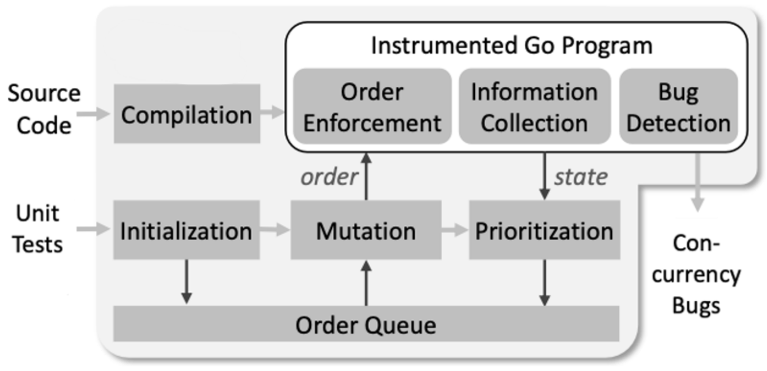


Figure 2.1: Overview of GFuzz's workflow[24]

As shown in Figure 2.1, GFuzz first identifies concurrent messages within the program and instruments the code with timer to establish message priorities. Specifically, GFuzz creates a scenario where a particular case is allowed to execute exclusively for a fixed period. Only if the case remains unsatisfied after the timeout does the execution return to a state where all cases have equal priority. Throughout each execution, GFuzz records the states and sequence of operations on channels to track program execution. Subsequently, GFuzz assesses whether block is constant on blocking goroutines based on the collected information. For instance, if a goroutine is blocked on a certain channel, GFuzz finds other goroutines with references to that channel and examines their operations on it to determine whether unblocking is possible, aiming to decide whether concurrency bugs exist. It then applies fuzzing techniques to modify the executed orders, creating order mutations. Simultaneously, GFuzz leverage the execution feedback to prioritize interesting orders that are more likely to trigger bugs, speeding up the bugs exposing process.

2.3.3 GCatch

GCatch[25] belongs to static detectors, which means it identifies bugs by analyzing code without running it. It consists of two main components: the blocking misuse of channel (BMOC) detector and the traditional detector. In the traditional detector, GCatch focuses on identifying traditional deadlocks caused by incorrect usage of mutex. It uses detection approaches employed in traditional programming languages[7], implementing an intra-procedural algorithm to pinpoint issues like double locks and conflicting locks that lead to resource deadlocks.

Algorithm 1 BMOC Bug Detection in GCatch[25]

```
1:  procedure BMOC DETECTOR($inputGoporgramP$)
2:      $C_{graph} \leftarrow$ ConstructCallGraph(P)
3:      $Primitives \leftarrow$ SearchPrimitives($C_{graph}$)
4:      $OP_{map} \leftarrow$ SearchOperations($Primitives, C_{graph}$)
5:      $D_{graph} \leftarrow$ ConstructDependGraph($OP_{map}, C_{graph}$)
6:      for each channel $c$ in $Primitives$ do
7:          $scope, P_{set} \leftarrow$ DisentanglingAnalysis($c, OP_{map}, D_{graph}$)
8:          $GO_{set} \leftarrow$ SearchGoroutines($c, scope$)
9:          $EPs \leftarrow$ ComputeExecutionPaths($GO_{set}, P_{set}, OP_{map}, scope$)
10:         for each execution paths $ep$ in $EPs$ do
11:             $Groups \leftarrow$ ComputeSuspiciousOperationGroups($ep, P_{set}, GO_{set}$)
12:             for each group ($\delta$) in $Groups$ do
13:                 $\Phi \leftarrow$ FilterConstraints($\delta$)
14:                 if Z3 finds a solution ($s$) for $\Phi$ then
15:                     ReportBug($\delta, s$)
```

Regarding the BMOC detector, GCatch utilizes the enumeration of all possible execution paths of goroutines within a specific group to analyze the code as shown in Algorithm 1. Initially, GCatch traverses all concurrency primitives and constructs a dependency graph that links them with their corresponding operations. Then, it creates sets of goroutines to group together the goroutines that access the same channel. During this process, GCatch disentangles the program to avoid the need to analyze the entire program for each group, focusing only on the primitives in the scope related to that channel. GCatch achieves this by finding the lowest common ancestor(LCA) of operations that are relevant to that channel in the dependency graph, thus generating the scope from the LCA operation. Subsequently, GCatch explores the execution paths within each goroutine set. It applies constraint conditions to filter out execution paths that are infeasible or containing no blocking operations, thus acquiring potential paths that could result in blocking. Finally, GCatch employs Z3[4] to verify whether these execution paths can lead to permanent blocking of goroutines, thereby detecting blocking bugs.

2.3.4 GoAT

For a given program, GoAT[29] set dynamic tracing as its first step. It instruments the program by spawning a goroutine to monitor its runtime behavior and injecting a handler before each concurrency primitive. These handlers disrupt the goroutines execution schedule by invoking *runtime.GoSched()*[14] method, achieving a broader range of execution interleavings[1]. During runtime, each operation on a primitive generates an event, which is recorded sequentially. These recorded events forms Execution Concurrency Trace (ECT), serving as the foundation for bug analysis.

Secondly, GoAT conducts offline analysis of program execution based on the ECT. It begins by constructing a goroutine tree, where edges show parent-child relationships between goroutines. The tool then traverses the goroutines to determine whether all of them successfully end after program termination. In the case of a goroutine leak, GoAT initially verifies the status of the root node goroutine (main goroutine). If it does not terminate as expected, GoAT reports a global deadlock. Subsequently, a breadth first search of the tree is performed. If a goroutine is found not to have terminated successfully, GoAT reports a partial deadlock and identifies the concurrency primitive responsible for the deadlock using the ECT.

Furthermore, GoAT models *Mutex*, *WaitGroup*, and other concurrent components based on the recorded events. This analysis determines whether these components operate as expected. For instance, it examines whether the counter for goroutines in a *WaitGroup* reaches zero upon program completion, enabling the identification of potential blocking bugs[15].

Chapter 3

Experiment

We conduct an effectiveness evaluation of four Go concurrency bug detection tools using the GoBench[33] dataset. To create a controlled environment and avoid hardware differences affecting tool performance, we perform the experiments on four identical servers with the following configurations: Intel(R) Xeon(R) CPU (4 total cores with 16 GB memory), 50GB disk, Ubuntu 20.04.6, and Docker version 24.0.4.

3.1 Research Question

In designing the experiments, we formulate the evaluation around the three following research questions:

RQ1. Installation and Execution Difficulty: *How easy is it to install and run the tools?*

To answer this question, we evaluate the installation and execution complexities associated with each of the four tools. For installation difficulty, we examine three aspects: the ease of following installation instructions, the compatibility with the latest Go versions, and the number of prerequisite tools. Regarding execution difficulty, we evaluate whether the bug detection process is automated, how the target code is stored, and whether the tool supports Docker. This assessment of installation and execution difficulty can serve as guidance for developers and researchers in selecting the most suitable tool for their specific requirements.

RQ2. Bug Detection Ability: *How effective are the tools in uncovering bugs in the dataset?*

To answer this question, we run each of the four tools on the GoBench dataset. To ensure a robust evaluation, we set the frequency of executions to 100 for Goleak and GoAT and run GFuzz on the entire dataset for 10 hours. As for the static analysis tool, GCatch, we perform three independent runs to verify result consistency. We categorize and analyse the detection results for both blocking and non-blocking bugs. Additionally, we classify the detected bugs based on their root causes, providing a more comprehensive understanding of the tools' performance for different bug categories. By providing an evaluation of the tools' bug detection abilities, our study equips developers and researchers with knowledge to choose suitable tool for different codes based on the concurrency primitives used.

RQ3. Execution Time and Overhead: *What are the tools' execution times, CPU usage and disk space usage?*

To answer this question, we measure the time taken by each tool to expose bugs in the dataset. Moreover, given the consistency in hardware configuration across the servers, we evaluate the CPU usage during tool execution and the disk space occupation after execution to assess the overhead caused by the tools. This assessment provides insights into the tools' efficiency and resource consumption, allowing developers and researchers to make informed decisions when selecting a bug detection tool based on their hardware and time constraints.

3.2 Dataset

GoBench[33], a comprehensive benchmark suite for Go concurrency bugs. It comprises a total of 185 bugs, categorized into two subsets: GoReal and GoKer. The GoReal subset includes 82 real-world concurrency bugs, while the GoKer subset comprises 103 kernel bugs. These kernel bugs are carefully extracted and simplified from 9 popular widely used open-source applications and recent research on Go concurrency bugs[31]. The extraction process focuses on preserving the complexity of the bugs while eliminating any irrelevant code.

We choose to use the GoKer dataset for our experiments as the extracted codes allow us to assess the tools' performance across various bugs more accurately, enabling a deeper analysis of differences in tool behaviours. In addition, the dataset includes various concurrency issues involving resource locks, channels, data races and anonymous functions, etc. By evaluating tools' performance on this comprehensive dataset, we can obtain reliable results that shows the capabilities of each tool in detecting different types of concurrency bugs. The GoReal dataset records the location of bugs within source codes and the corresponding Docker file to reproduce the execution of these bugs.

However, instead of providing the isolated buggy code like GoKer, which would allow for more focused testing, this format makes it challenging to test target bugs, potentially impacting the accuracy of our test results. Additionally, conducting tests on a large code hinders our ability to perform in-depth result analysis. As a result, we choose not to use samples from the GoReal dataset. It is worth noting that all bug types present in GoReal are included by the GoKer dataset. Therefore, the comprehensiveness of our results remains unaffected, ensuring a robust evaluation of the tools' performance.

Chapter 4

Result

In this chapter, we investigate the outcomes derived from our experiments, which align with the research questions formulated in the preceding Experiment chapter.

4.1 RQ1: Installation and Execution Difficulty

To assess the installation and execution difficulty for each of the four tools, we categorized the levels of difficulty as *Easy*, *Medium*, and *Hard*. As previously mentioned, we conduct a comprehensive evaluation of both installation and execution complexities. The results of this assessment are summarized in Table 4.1. Regarding installation complexity, if a tool meets the criteria of *Ease of following instructions* and *Go versions compatibility*, and requires no more than 1 prerequisite tool, we categorize it as *Easy*. If either criterion is not met, it falls under *Medium*. If neither criterion is met, it is labeled as *Hard*. The same principles apply to the three indicators of execution difficulty. It's worth noting that Goleak falls into the *Hard* category for execution difficulty due to its limited automation in detection, even if it meets *Unconstrained code storage*. We provide detailed explanation on the results in the following sections.

4.1.1 GFuzz

GFuzz exhibits excellent user-friendliness during the installation process. Following the provided instructions allows for a seamless and successful installation. The instructions also comprehensively outlines the tool's command usages and corresponding examples, ensuring a clear path for users. Moreover, GFuzz's compatibility with various Go versions is good, as the installation shell script specifies the versions of requisite Go

Tools	Installation	Execution	Ease of following instructions	Go versions compatibility	Prerequisite tools number	Automated detection	Unconstrained code storage	Docker supported
GFuzz	Easy	Easy	✓	✓	1	✓	✓	✓
GCatch	Medium	Medium	×	✓	4	✓	×	×
Goleak	Easy	Hard	✓	✓	1	×	✓	×
GoAT	Hard	Medium	×	×	2	✓	×	×

Table 4.1: Installation and Execution complexity evaluation results

environment and dependency packages clearly. Notably, Docker is the only prerequisite of the installation process, which contributes to the ease of installation.

In terms of execution, GFuzz's simplifies the testing process by automatically detecting bugs in the whole application without having to manually modify any source code. Furthermore, the inclusion of Docker support prevents any potential interference with other tools or applications, enhancing its usability. Flexibility in the storage of target code, coupled with the ability to get target codes using Git URL, provides users with convenience in their testing process. The overall experience with GFuzz highlights its accessibility and testing automation design, promoting an efficient bug detection workflow.

4.1.2 GCatch

The installation of GCatch presents some challenges due to certain complexity in the process. Following the instructions poses difficulties because the absence of explicit guidance on prerequisite tools and potential issues related to modifying the *GOPATH* environment variable, which leads to continuous installation failures. Additionally, the tool's code is required to be in Go environment directory, which complicates each subsequent use and hinders accessibility, especially for the users without root privilege. The compatibility of GCatch with latest Go versions is evident, facilitated by its usage of Go modules. GCatch has the largest number of prerequisite tools, including G++, Python, Make, and the Go environment, but this does not add much complexity because these tools are not difficult to install.

Upon execution, GCatch's bug detection process also is automated, making it a convenient choice for testing applications composed by massive code files. However, the lack of Docker support introduces potential conflicts with other tools, limiting its flexibility within a shared environment. Furthermore, GCatch's requirement for the target code to reside within Go environment introduces constraints on execution, limiting its applicability for users without enough privileges like its installation process. While the overall experience with GCatch exhibits a balance between installation challenges and bug detection efficiency, there are limitations in code storage and Docker support that need to be considered in specific testing scenarios.

4.1.3 Goleak

Goleak's installation process proves to be easy and accessible. The provided instructions facilitate a straightforward installation experience as no additional steps beyond package downloading are required. Compatibility with the latest Go versions is ensured by the continuous update of the Goleak package. Additionally, For old versions of the Go environment, we can still find corresponding versions in the historical releases of Goleak. Furthermore, Go environment is the only prerequisite for the installation process.

However, in terms of execution, Goleak presents challenges. The absence of an automated bug detection process requires manual code insertion to call functions of Goleak, which implies that using Goleak in real-world Go applications involves making numerous manual changes to the source code and running it multiple times. Although the tool lacks Docker support, it doesn't impact the local Go environment, so it won't interfere with other tools. Additionally, because Goleak is invoked directly through its package, the target code can be placed in any location with a Go environment. Although Goleak's straightforward installation and compatibility with various Go versions remain valuable aspects of its usability, the manual intervention required for code insertion introduces critical complexity, which increases with code scale.

4.1.4 GoAT

The installation process of GoAT poses considerable challenges, reflecting its complexity. The need for a dual Go environment setup and the multiple manipulation of the global Go environment using link file command result in a complex installation procedure. Moreover, the requirement for additional environment variables to specify result storage locations further adds to the complexity. The installation instructions face issues, as the provided commands cannot be used after installation. In our experiments, we opt to directly invoke the executable binary file after the Go build process. Compatibility with the latest Go versions is poor due to GoAT's lack of Go module integration, necessitating manual package installation to address deprecated functions and other version-specific issues. Go and Make are the only prerequisite tools for GoAT, which alleviates some installation challenges.

Upon execution, GoAT's automated bug detection process eases the testing process by eliminating the need for manual code alterations before testing. However, the absence of Docker support and its environment-switching nature raise concerns about potential conflicts with other Go programs on the same machine. The flexibility in target code storage provides adaptability in execution, but an additional configuration file is required to specify the code's location.

4.1.5 Summary

The results of the installation and execution difficulty assessment shed light on the accessibility, user-friendliness, and practicality of each tool. GFuzz stands out for its easy installation and clear instructions. Its automated bug detection process and Docker support enhance usability in execution. GCatch and Goleak presents moderate installation and execution challenge. Conversely, GoAT's installation complexity is notable due to a dual Go environment setup and the lack of compatibility with different Go versions, with execution offering an automated bug detection process but posing concerns about Docker support and environment-switching effects. These insights provide developers and researchers with valuable guidance in selecting appropriate tools based on their specific needs and constraints.

4.2 RQ2: Bug Detection Ability

To investigate the bug detection capabilities of the four tools, we apply them to the task of detecting 103 bugs from the GoKer dataset. As shown in Table 4.2, we present a summary of the detection outcomes[1] and organize them into categories based on the causes of the bugs, offering a clear comparison.

[1]The full outputs can be found in Appendix

Bug Type			Tools			
Category	Cause	Subcause(#Num)	Goleak	GCatch	GFuzz	GoAT
Blocking	Resource Deadlock	AB-BA deadlock(6)	2	3	0	4
		Double locking(12)	11	12	0	11
		RWR deadlock(5)	1	3	0	2
	Communication Deadlock	Channel(17)	15	7	8	13
		Channel & Condition Variable(2)	2	0	1	0
		Channel & Context(8)	6	2	5	6
		Condition Variable(2)	2	0	0	2
	Mixed Deadlock	Channel & Lock(13)	7	5	4	5
		Channel & WaitGroup(2)	1	0	0	2
		Misuse WaitGroup(1)	1	0	0	1
Nonblocking	Go-Specific	Anonymous function(4)	3	0	0	1
		Misuse channel(6)	6	1	1	1
		Testing library(2)	0	0	0	0
		WaitGroup(2)	1	0	0	1
	Traditional	Data race(20)	18	2	2	3
		Order violation(1)	1	1	1	1

Table 4.2: Bug detection results on GoKer for the tools

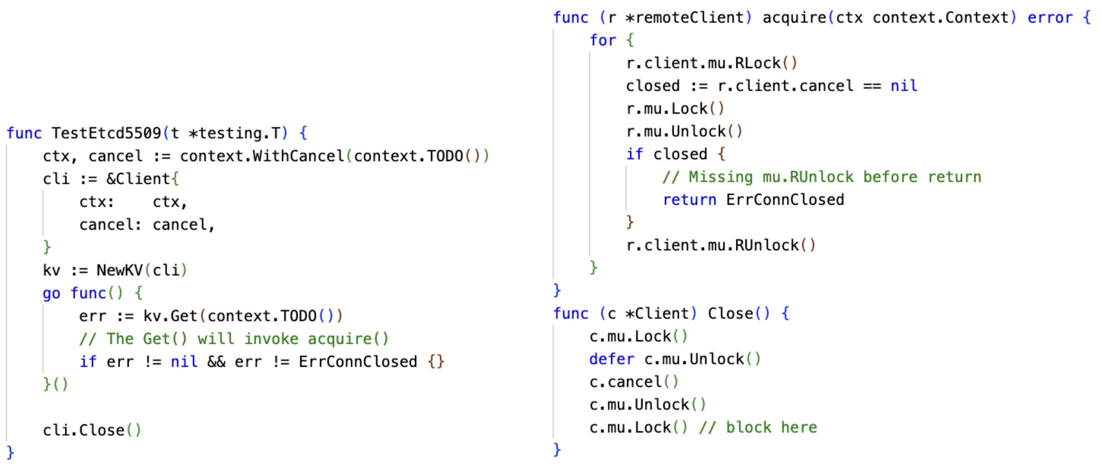
4.2.1 Blocking Bugs

4.2.1.1 Resource Deadlock

Within the Resource Deadlock category, a total of 23 bugs are identified. Notably, GCatch exhibits the highest bug detection count, containing 18 instances. Moreover, GCatch's results mostly accurately identify the causes of bugs, such as correctly labeling *kubernetes_30872* as Conflict Lock and identifying *etcd_6708* as Double Lock. This can be attributed to GCatch's nature as a static detector, embedded with static checkers developed for traditional programming language bugs, which encompass the bugs present within the Resource Deadlock category. GoAT follows closely with 17 instances, while Goleak identifies 14 bugs. It is worth noting that GFuzz yields no bug detection in this category. This is attributed to GFuzz's utilization of the Message Reordering technique, which exclusively targets bugs originating from concurrent messages. As a result, bugs triggered by resource locks remain outside GFuzz's detection scope. Therefore, we will not evaluate GFuzz's performance within this section. Among the three remaining tools, though the detection counts are relatively close, differences emerge in the specific bugs detected.

AB-BA deadlock. For the case of AB-BA deadlocks, a comparable number of bugs are identified across the three tools, but each tool gets distinct outcomes. Interestingly, there is no overlap in the bugs detected by all three tools, nor are there any instances that go unnoticed by all tools. Although the performance of all three tools in this bug category is not optimal, the combined use of GoAT and GCatch still successfully exposes all bug samples within this type.

Double locking. All three tools demonstrate strong performance on the bug samples within this category. GCatch successfully detects all bugs, while Goleak and GFuzz both miss the bug sample *etcd_5509*. As the code shown in Figure 4.1, we find that the bug in *etcd_5509* will not be triggered under normal executions. This is due to the fact that the cancel function of *context.TODO()*, obtained through *WithCancel*, returns a non-nil function[9]. Consequently, the *closed* variable in the *acquire* function remains false, rendering the bug inactive, even though *RUnlock* is absent if *closed* becomes true. Thus, this particular bug goes undetected by Goleak and GFuzz, as the buggy code block won't be executed in this scenario. However, GCatch includes this case in the execution paths computed in static analysis, despite it being unattainable in practice due to the non-nil cancel function of *context.TODO()*.



```go
func TestEtcd5509(t *testing.T) {
    ctx, cancel := context.WithCancel(context.TODO())
    cli := &Client{
        ctx:    ctx,
        cancel: cancel,
    }
    kv := NewKV(cli)
    go func() {
        err := kv.Get(context.TODO())
        // The Get() will invoke acquire()
        if err != nil && err != ErrConnClosed {}
    }()

    cli.Close()
}
```

```go
func (r *remoteClient) acquire(ctx context.Context) error {
    for {
        r.client.mu.RLock()
        closed := r.client.cancel == nil
        r.mu.Lock()
        r.mu.Unlock()
        if closed {
            // Missing mu.RUnlock before return
            return ErrConnClosed
        }
        r.client.mu.RUnlock()
    }
}
func (c *Client) Close() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.cancel()
    c.mu.Unlock()
    c.mu.Lock() // block here
}
```

Figure 4.1: Bug *etcd_5509* caused by double locking. (Code simplified for explanation)

RWR deadlock. In this category, GCatch performs more effectively, revealing the highest number of bugs. We observe that the bug *cockroach_16167* isn't exposed by any tool. Upon analyzing the code, we find that GCatch's failure to uncover this bug is due to its lack of information about the function *sync.NewCond*, which leads to it not recognizing that *systemConfigCond* and *systemConfigMu* are the same mutex lock. In addition, the bug sample retains only the portion of code related to the locks during kernel extraction, causing consecutive *Lock()* and *Unlock()* operations on *systemConfigMu*. This arrangement makes it challenging for another goroutine to insert an *RLock()* operation in between, causing both dynamic detectors, Goleak and GoAT, to fail exposing this bug. A similar situation exists in *kubernetes_62464*, where the simplified operations between lock and unlock methods make exposing the deadlock more difficult for the dynamic tools. However, GCatch exposes this bug successfully as no equivalent lock present in the code. This demonstrates the advantage of static analysis strategies in detecting bugs that might not easily emerge in normal execution scenarios.

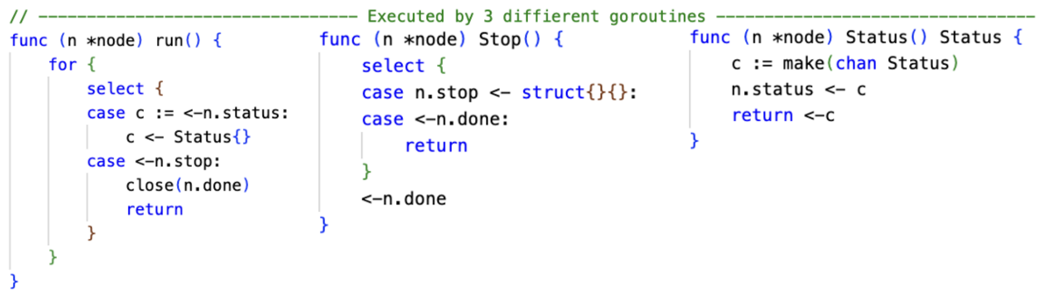
```
systemConfigCond = sync.NewCond(systemConfigMu.Locker())
G1                          G2
e.Start()
e.updateSystemConfig()
                            e.execParsed()
                            e.systemConfigCond.L.Lock()
e.systemConfigMu.Lock()
                            e.systemConfigMu.RLock()
----------------------G1,G2 deadlock--------------------
```

Figure 4.2: The steps to trigger the deadlock in *cockroach_16167*

4.2.1.2 Communication Deadlock

Communication Deadlock refers to the scenario where goroutines wait for message resources. WaitGroup and Condition Variable represent traditional communication messages, while messages related to channels are Go-Specific. A total of 29 bugs fall into this category. Notably, Goleak performs the best, detecting 25 bugs. On the other hand, GCatch exhibits poorer performance in this bug category, exposing only 9 bugs. Although GFuzz exposes fewer bugs compared to Goleak and GoAT, with only 14 bugs revealed, upon analyzing the code, we discover that GFuzz uncovers almost all bugs related to the *select-case* primitive. Some of these bugs are missed by Goleak or GoAT, such as *etcd_6857* and *istio_17860*. We will deliver a detailed discussion of these findings in the subsequent sections.

Channel. For bugs only related to channels, Goleak often succeeds in exposing them, as detecting blocked goroutines for blocking bugs is a straightforward and effective approach. As we discussed previously in the Resource Deadlock part, Goleak's weakness lies in its inability to expose bugs when their occurrence is dependent on rare conditions due to the lack of enforced execution order. However, in the case of channel-related bugs, since messages involve modifications beyond just sending and receiving, there are more possibilities within the execution path at runtime. However, such instances can still be found. Only GCatch and GFuzz exposed *etcd_6857*. As shown in Figure 4.3, this bug involves three goroutines that run sequentially: one for transmitting data status, one for receiving status and stopping upon receiving stop message and the last one for transmitting stop message. When the third goroutine executes prematurely, i.e., *Stop()* occurs before status transmission, the goroutine responsible for receiving data status becomes blocked. As shown in Figure 4.4, GCatch detects the bug after analyzing the computation path of the channel for transmitting data status, while GFuzz identifies the bug by reordering the cases in *run()* method and exposing it in the case where message from *n.stop* is received first. Goleak and GoAT, on the other hand, fail to expose this bug due to the relative simplicity of the status transmission, making it hard for *Stop()* to be executed first.

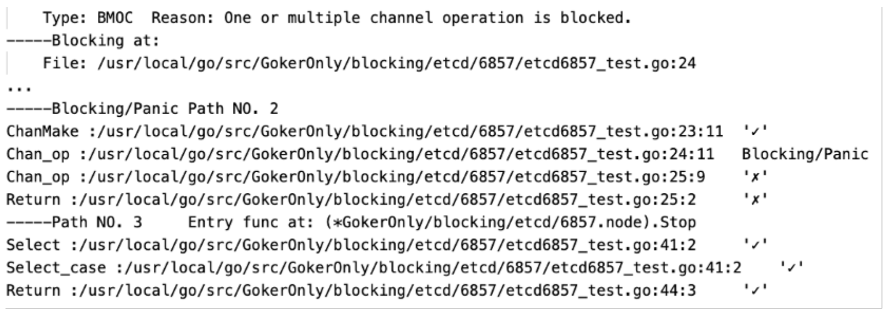


```
// ---------------------------------- Executed by 3 diffierent goroutines ----------------------------------
func (n *node) run() {
    for {
        select {
        case c := <-n.status:
            c <- Status{}
        case <-n.stop:
            close(n.done)
            return
        }
    }
}

func (n *node) Stop() {
    select {
    case n.stop <- struct{}{}:
    case <-n.done:
        return
    }
    <-n.done
}

func (n *node) Status() Status {
    c := make(chan Status)
    n.status <- c
    return <-c
}
```

Figure 4.3: The functions for goroutines in *etcd_6857*

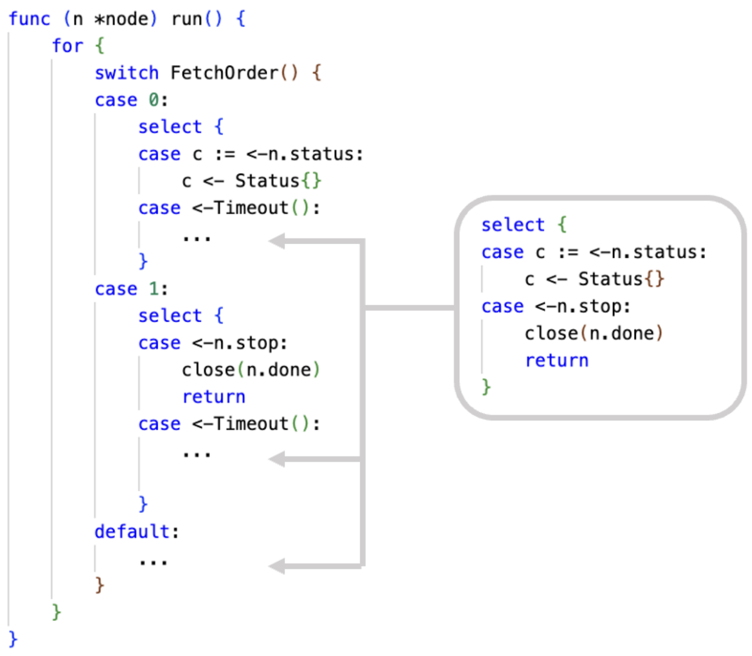
Condition variable. In this section, we concurrently address both **Condition variable** and **Channel & Condition Variable** types of bugs, due to the limited number of samples in this category and the fact that only the bug in *moby_27782* among the samples in the latter category involves channels. In *moby_27782*, two goroutines become blocked — one waiting for the other to modify the condition variable, and the other waiting for message from the former through a channel, which results in a deadlock. Although *kubernetes_11298* also employs channels, only condition variable contributes to the occurrence of the bug. Goleak detects all four bugs in this type, showcasing its robust performance in this bug type. GFuzz successfully exposes *moby_27782*, which involves the usage of *select* for managing various operations on the channel.

Channel & Context. While samples in this category involve the use of the context package, upon examining the code, we find that the cause of bug occurrence is like



```
|   Type: BMOC  Reason: One or multiple channel operation is blocked.
-----Blocking at:
|   File: /usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:24
...
-----Blocking/Panic Path NO. 2
ChanMake :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:23:11  '✓'
Chan_op :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:24:11   Blocking/Panic
Chan_op :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:25:9    '✗'
Return :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:25:2     '✗'
-----Path NO. 3     Entry func at: (*GokerOnly/blocking/etcd/6857.node).Stop
Select :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:41:2     '✓'
Select_case :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:41:2    '✓'
Return :/usr/local/go/src/GokerOnly/blocking/etcd/6857/etcd6857_test.go:44:3     '✓'
```

(a) GCatch's bug report for *etcd_6857*



```
func (n *node) run() {
    for {
        switch FetchOrder() {
        case 0:
            select {
            case c := <-n.status:
                c <- Status{}
            case <-Timeout():
                ...
            }
        case 1:
            select {
            case <-n.stop:
                close(n.done)
                return
            case <-Timeout():
                ...
            }
        default:
            ...
        }
    }
}
```

```
select {
case c := <-n.status:
    c <- Status{}
case <-n.stop:
    close(n.done)
    return
}
```

(b) GFuzz's order enforcement in the *run()* method

Figure 4.4: Illustration of the tools' operations on bug *etcd_6857*

that of samples where the cause is a channel. In other words, bugs still come from improper channel send or receive operations. This is because the *Context* usage focus on managing and controlling goroutines[9], such as transmitting termination signals by combining the cancel function and the *ctx.Done()* method, leading to changes in conditions following the *case* statements. However, the formation of concurrency bugs remains centered around the usage of channels. Therefore, detection results resemble those in the Channel category. Goleak and GoAT expose the highest number of bugs, totaling 6. GFuzz identifies some less easily exposed bugs, such as *istio_18454*.

4.2.1.3 Mixed Deadlock

Mixed Deadlock refers to deadlocks caused by a combination of resource locks and message passing. Across this category's 16 bug samples, the performance of the four tools is less than optimal. Goleak and GoAT show close performance, detecting 9 and 8 bugs respectively. In contrast, GCatch and GFuzz detect only 5 and 4 bugs. The performance of GCatch and GFuzz aligns with expectations, because GFuzz exclusively targets bugs related to concurrency messages and GCatch's static analysis of concurrency primitives, such as resource locks and channels, operates independently, which resembles a union of multiple checkers. Therefore, when these two causes are combined, GCatch struggles to expose most of these bugs.

Channel & Lock. The performance of the four tools on the samples in this category all are not satisfactory. Among the 13 bug samples in this category, Goleak still performs the best, exposing 7 bugs. The good news is that by employing all four detection tools simultaneously to tackle these mixed cause bugs, only 3 bugs are left undetected. However, this also warrants our attention as these bugs concurrently expose the deficiencies of all four tools. For instance, consider *etcd_6873*, where the main goroutine G1 generates data and creates another goroutine G2 to perform coalesce operations. Simultaneously, a third goroutine G3 is responsible for closing the channel used in the process. The latter two goroutines ensure mutual exclusion through a mutex. However, G2, responsible for data coalescing, is also in charge of closing the *donec* channel used for receiving data in G3. Therefore, if G3 gains control of the mutex first, it will be blocked by *donec*, causing G2 to be blocked by the *wbs.mu* lock, as shown in Figure 4.5. In this bug, since G2 immediately performs coalescing upon receiving data from *updatec* and anticipate acquiring the *wbs.mu*, it becomes challenging for G3 to preemptively acquire the mutex. As a result, both Goleak and GoAT fail to detect this bug during dynamic detection. GFuzz also fails to enforce reordering the execution path because the order is unaffected by the *select-case* keyword. GCatch, on the other hand, is unable to analyze the buggy path due to the different types of primitives that block G2 and G3 in the deadlock. It can be inferred that the mixed causes for bugs introduce higher complexity in detection. The combined use of multiple tools becomes an option, taking advantage of their different strengths.

WaitGroup Related. This category encompasses two causes: **Channel & WaitGroup** and **Misuse WaitGroup**. All bug samples of this type are successfully detected by GoAT. GoAT optimizes its detection for the *WaitGroup* primitive by recording related

```
 G1                           G2                    G3
 newWatchBroadcasts() //G2
 wbs.update()
 wbs.updatec <-
 return
                              <- wbs.updatec
                              wbs.coalesce()
                                                   wbs.stop()
                                                   wbs.mu.Lock()
                                                   close(wbs.updatec)
                                                   <- wbs.donec
                              wbs.mu.Lock()
-------------------------G2,G3 deadlock---------------------------
 Operations Left:             wbs.mu.Unock()       wbs.mu.Unlock()
                              close(wbs.donec)
```

Figure 4.5: The steps to trigger the deadlock in *etcd_6873*

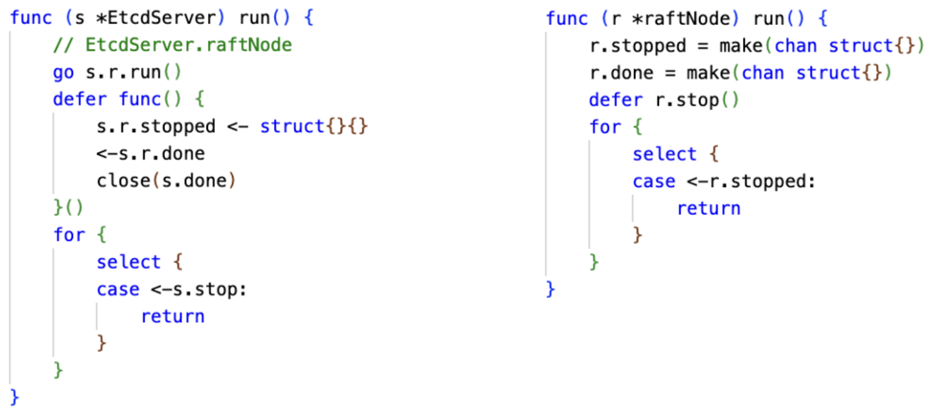
events. Additionally, because *WaitGroup* is essentially a goroutine counter[15], it is less affected by execution order compared to locks, because the increment and decrement of counters do not involve as many scenarios as the locking and unlocking of read-write locks. When dealing with bugs related to *WaitGroup*, GoAT stands as the preferred choice due to its effective optimization and the relative simplicity of the *WaitGroup* primitive.

4.2.2 Nonblocking Bugs

There are 35 nonblocking bug samples in the GoKer dataset, out of which Goleak detects 29. We observe that the remaining three tools detect no more than 7 bugs each. This limitation is attributed to their primary focus is blocking concurrency bugs, rendering them incapable of identifying nonblocking bugs. Some bugs in the results are detected by these three tools due to their potential for causing blocking behavior. For example, in the case of *kubernetes_88331*, there exists a risk of a goroutine getting blocked due to channel *stopCh* at line 91, rather than just the risk of triggering a data race. As a result, we exclusively focus on discussing the performance of Goleak in this section.

4.2.2.1 Go-Specific

Goleak successfully detects 10 out of the 14 bugs in the Go-Specific category. For bugs falling into the Misuse channel category, the causes primarily are sending messages to a closed channel. For instance, in the case of , if a message is sent to the *r.stopped* channel by *EtcdServer* before the execution of the *run()* method of *raftNode*, a bug can arise because the *stopped* channel is not initialized yet. Goleak successfully detected all the bugs within the Misuse channel subcategory. This is because sending messages to an uninitialized channel triggers a panic in the Go environment[13]. which can be leveraged to find the goroutine that caused it and trace back its stack, thus making it relatively easy for Goleak to identify and locate the bugs of this type.



```go
func (s *EtcdServer) run() {
    // EtcdServer.raftNode
    go s.r.run()
    defer func() {
        s.r.stopped <- struct{}{}
        <-s.r.done
        close(s.done)
    }()
    for {
        select {
        case <-s.stop:
            return
        }
    }
}
```

```go
func (r *raftNode) run() {
    r.stopped = make(chan struct{})
    r.done = make(chan struct{})
    defer r.stop()
    for {
        select {
        case <-r.stopped:
            return
        }
    }
}
```

Figure 4.6: The functions involved in misusing channel within *etcd_3077*.

For the remaining subcategories, Goleak's detection method basically revolves around spotting data races. Take the example of *cockroach_35501*, where the developer invokes the method *validateCheckInTxn()* within an anonymous function, which carries a risk of data race. Goleak detected this bug because its data race detection is focused on memory access. Thus, the nested structure of anonymous functions doesn't interfere with the detection process. However, for bugs not fitting this pattern, Goleak lacks the capability to detect them. For instance, in the case of *serving_6171*, the issue comes from the misuse of the Testing package[16], where the test doesn't wait for all goroutines to finish before concluding. This can lead to unpredictable and inconsistent test results, resulting in test flakiness. Because this bug originates from the improper usage of the Testing package, it goes undetected by Goleak.

4.2.2.2 Traditional

For the traditional nonblocking bugs, Goleak successfully detects 19 of the 21 samples. Among these samples, 20 bugs are attributed to data race, which stands as a main cause of nonblocking bugs. Goleak's detection mechanism for data race is effective as it detects 18 out of 20 bugs in this category. In the Order violation category, the only sample, *moby_18412*, presents a unique scenario. In this case, the *RunCommand-WithOutputForDuration()* method spawns a goroutine that can modify one of its return values, aiming to influence the return only when it triggers the function to return. But it might also impact the return value when the function is returning due to the timeout. As shown in Figure 4.7, when the process is killed due to the timeout, the *Wait()* method completes[11] and the waiting goroutine could set *exitCode* to 1 before the function returns, which is not an intended behavior. Goleak identifies this race condition by detecting that the main goroutine and the waiting goroutine may access the *exitCode* concurrently, with the waiting goroutine performing a write operation. We observe that the other three tools detect this bug as a partial deadlock, as the waiting goroutine also sends message to the *done* channel. However, in the case of timeout, this message will not be received by the main goroutine, causing the waiting goroutine to become blocked. The result shows Goleak's reliability in dealing with traditional nonblocking bugs, making it a good choice for detecting this type of bugs.

```go
go func() {
    // Wait for cmd to exit in the goroutine
    exitErr := cmd.Wait()
    // One of the return variables, default is 0
    exitCode = 1
    done <- exitErr
}()

select {
case <-time.After(duration):
    killErr := cmd.Process.Kill()
    if killErr != nil {
        fmt.Printf("failed to kill (pid=%d): %v\n", cmd.Process.Pid, killErr)
    }
    timedOut = true
    break
case err = <-done:
    break
}
return
```

Figure 4.7: The buggy code block in *moby_18412*.

4.2.3 Summary

Blocking bugs. Each of the four tools has shown different strengths and performances:

GCatch emerges as a reliable detector in the domain of Resource Deadlock. Leveraging its static analysis approach, GCatch could uncover bugs that might be challenging to trigger in the actual runtime. Additionally, GCatch optimizes its resource lock analysis by embedding traditional checkers, which has demonstrated its practical value. However, GCatch's performance fails in the domain of Mixed Deadlock, where the independence of checkers for channels and locks renders it less effective at exposing bugs from mixed causes. Furthermore, when dealing with equivalent primitives introduced by library functions, GCatch struggles due to the lack of function information.

GFuzz, designed exclusively for Communication Deadlock, employs message reordering technique to expose bugs related to message passing. Its effectiveness in this domain is evident, with the capability to uncover a significant portion of bugs linked to *select-case* statement. The enforcement of message reordering reveals subtle bugs that might hide in unnoticed cases. However, GFuzz has limited detection ability for other bug types.

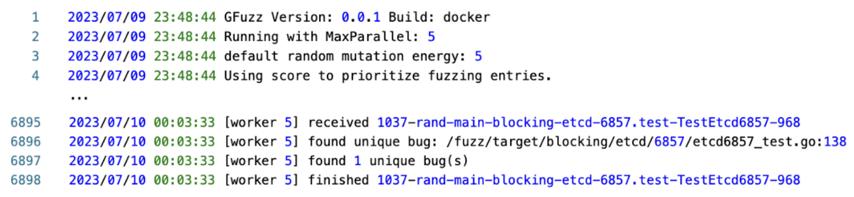
Goleak distinguishes itself through its comprehensive detection strategy, analyzing the state and stack trace of each goroutine. This universal approach ensures robust detection capabilities across various bug types. Goleak proves to be a good choice when the type of target bug is uncertain.

GoAT, as another dynamic detector, employs events analysis of different concurrency primitives to identify bugs. Its bug detection ability across most categories is close with those of Goleak. GoAT's strategy allows for optimization for specific primitives, yielding better results for primitives like *WaitGroup*.

Nonblocking bugs. Goleak stands as the only tool equipped with detection ability among the four tools for this bug type. It employs modules for identifying misuse of channels and data races, exhibiting high effectiveness. However, Goleak's performance in uncovering nonblocking bugs arising from other causes is less satisfactory.

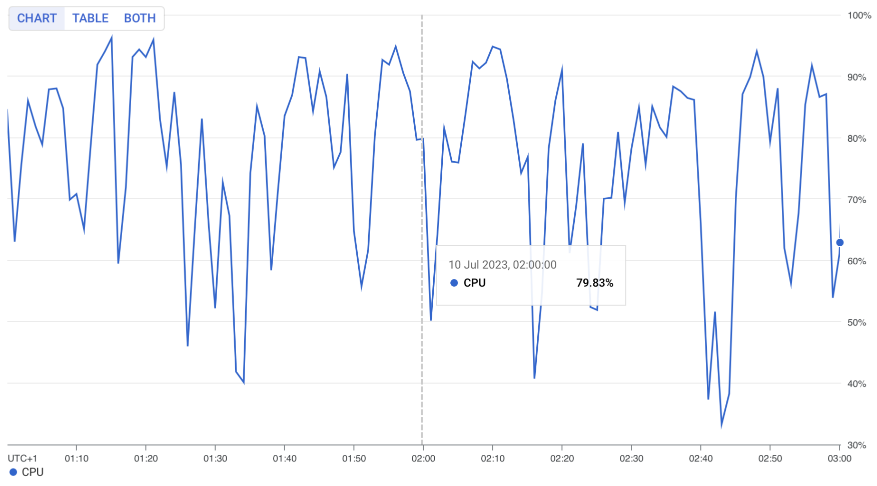
4.3 RQ3: Execution Time and Overhead

To study the execution time and resource overhead of the four tools, we collect data as they perform detection on GoKer, which is shown in Table 4.3. This includes tracking metrics such as disk usage, CPU usage, execution time and average time taken to expose a bug. Employing identical blank servers for experimentation allows us to determine disk usage by subtracting its pre-installation state from the disk occupancy after tool execution. This encompasses usage such as the execution environment, tools, code, detection logs, and intermediate results. CPU usage during tool execution is obtained using Google Cloud platform's CPU monitoring tool, and we subtract the usage of idling server to obtain final CPU usage. Execution time represents the time taken by the tools to complete detection on all GoKer samples. Notably, for GFuzz, we consider the time when it last detects a bug, even though we have run it for 10 hours, since GFuzz does not terminate autonomously and can run indefinitely without manual termination. In Figure 4.8, we present the data sources for GFuzz's CPU usage and execution time as an example.



```
2023/07/09 23:48:44 GFuzz Version: 0.0.1 Build: docker
2023/07/09 23:48:44 Running with MaxParallel: 5
2023/07/09 23:48:44 default random mutation energy: 5
2023/07/09 23:48:44 Using score to prioritize fuzzing entries.
...
2023/07/10 00:03:33 [worker 5] received 1037-rand-main-blocking-etcd-6857.test-TestEtcd6857-968
2023/07/10 00:03:33 [worker 5] found unique bug: /fuzz/target/blocking/etcd/6857/etcd6857_test.go:138
2023/07/10 00:03:33 [worker 5] found 1 unique bug(s)
2023/07/10 00:03:33 [worker 5] finished 1037-rand-main-blocking-etcd-6857.test-TestEtcd6857-968
```

(a) Timestamps in GFuzz's execution logs



(b) CPU monitor for GFuzz's execution server

Figure 4.8: Presentation of data sources for GFuzz in RQ3

Observing disk usage, GoAT shows the highest consumption at 2.7GB, followed by GFuzz at 1.85GB. GoAT's large disk usage stems from generating substantial intermediate files while tracing each primitive and the tool's complex execution environment, which accounts for nearly 1.7GB after installation. GFuzz's disk usage is mainly attributed to execution logs, encompassing results and configuration files. While individual log files are small, only around 10KB each, the cumulative effect of running for 10 hours generates nearly 87,000 logs. Comparatively, Goleak and GCatch show smaller disk consumption due to their avoidance of persistent storage of intermediate results during execution. In terms of CPU usage, GFuzz consumes the highest amount of resources due to deploying multiple workers to parallel detect different samples. In contrast, Goleak's CPU usage is minimal as its detection process mainly is monitoring

Tools	Disk Usage	CPU Usage	Time	Avg. Time
Gfuzz	1.85G	77.80%	14min49sec	40.41s
Gcatch	1.1G	55.52%	5min40sec	9.44s
GoAT	2.7G	70.12%	85min15sec	96.51s
Goleak	0.87G	29.42%	160min17sec	124.9s

Table 4.3: Comparison of results from different tools in RQ3

goroutine runtime stack within the program, which consumes relatively fewer resources.

Regarding runtime, GoAT and Goleak display notably longer duration, both in overall execution time and average time taken to expose a bug. This can be attributed to the fact that both GoAT and Goleak need to completely execute the code for each detection, and their global deadlock detection employs a timeout mechanism, where a program is classified as a global deadlock when its execution time exceeds threshold, incurring substantial time overhead. GCatch relies on static code analysis, offering the fastest execution time. Regarding GFuzz, despite being run for 10 hours, it reveals all 22 bugs in its result within 14 minutes and 49 seconds. Thus, we opt to consider this data as the actual execution time, showing a fast detection speed like GCatch, which is attributed to its multi-worker mechanism.

In summary, Goleak occupies the least system resources, but it has the longest runtime. Comparatively, GCatch achieves the fastest detection speed and utilizes relatively moderate resources. In cases of constrained CPU performance, it's recommended to avoid using GFuzz. Likewise, for systems with restricted disk space, caution is needed when choosing GoAT.

Chapter 5

Conclusion

5.1 Summary

In conclusion, this paper has presented a comprehensive investigation into the realm of Go concurrency bugs. By introducing the concept of Go concurrency bugs and utilizing the GoBench benchmark suite, encompassing various bug categories, we systematically evaluate four popular and leading Go concurrency bug detection tools. This evaluation covers several aspects like tool deployment difficulty, bug detection efficacy and runtime overhead, providing an insightful understanding of each tool's features. This study equips Go developers with valuable insights to guide their selection of bug detection tools and enhances the knowledge of the Go bug detection process. Summarizing the results in response to our research questions, we conclude that Goleak is the prime choice for nonblocking bugs. For blocking bugs, if the software size is small, Goleak is recommended; for larger software, GoAT is suitable for machine with higher performance, while GCatch is preferable with lower performance. A program with frequent resource lock usage favours GCatch, whereas a substantial usage of the *select-case* statements indicates GFuzz as an ideal solution.

5.2 Limitations and Future work

It is essential to acknowledge the limitations of our work. While using GoKer dataset for our experiments enables more accurate bug detection analysis and result evaluation, not conducting experiments on real-world large applications could introduce deviation between experiments and practical scenarios. This deviation might affect the tools' detection abilities. Additionally, the assessment of runtime resource consumption might be inaccurately represented due to the varying rates at which different tools experience increased overhead as code size grows. For instance, Goleak's accumulation of goroutine stack traces contributes to runtime overhead more than GCatch's localized analysis of each primitive's call graph. Looking forward, our future work aims to address these limitations by conducting experiments on larger, real-world software to validate the reliability and consistency of our evaluation results. Furthermore, we observed that in certain cases, such as Channel & Lock category bugs, none of the four tools demonstrates optimal performance. Therefore, we aspire to identify or design improved bug detection tools to deal with these cases, enhancing the reliability of Go bug detection methods.

Bibliography

[1] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News*, 38(1):167–178, 2010.

[2] Yan Cai and Qiong Lu. Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering*, 42(9):825–842, 2016.

[3] Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 474–489, 2022.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] Nicolas Dilley and Julien Lange. An empirical study of messaging passing concurrency in go projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 377–387. IEEE, 2019.

[6] Docker. Docker - build, ship, and run any app, anywhere. https://www.docker.com, 2023.

[7] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review*, 37(5):237–252, 2003.

[8] GitHub. Fastest growing languages. https://octoverse.github.com, 2023.

[9] Google. The context package in go. https://pkg.go.dev/context, 2023.

[10] Google. Effective go. https://go.dev/doc/effective_go, 2023.

[11] Google. The exec package in go. https://pkg.go.dev/os/exec, 2023.

[12] Google. The go programming language. https://go.dev, 2023.

[13] Google. The go programming language specification. https://go.dev/ref/spec, 2023.

[14] Google. The runtime package in go. https://pkg.go.dev/runtime, 2023.

[15] Google. The sync package in go. https://pkg.go.dev/sync, 2023.

[16] Google. The testing package in go. https://pkg.go.dev/testing, 2023.

[17] Omar Inverso, Truc L Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 807–812. IEEE, 2015.

[18] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th joint meeting of the European software engineering conference and the acm sigsoft symposium on the foundations of software engineering*, pages 13–22, 2009.

[19] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for c/pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 379–390, 2016.

[20] Kubernetes. Kubernetes - production-grade container orchestration. https://kubernetes.io, 2023.

[21] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: Liveness and safety for channel-based programming. *ACM SIGPLAN Notices*, 52(1):748–761, 2017.

[22] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1137–1148, 2018.

[23] Stanley B. Lippman, Jose Lajoie, and Barbara E. Moo. *C++ Primer*. Addison-Wesley Professional, 5th edition, 2012.

[24] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. Who goes first? detecting go concurrency bugs via message reordering. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 888–902, 2022.

[25] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in go software systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 616–629, 2021.

[26] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[27] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 174–184, 2016.

[28] Jennfer B Sartor and Lieven Eeckhout. Exploring multi-threaded java application performance on multicore hardware. *ACM SIGPLAN Notices*, 47(10):281–296, 2012.

[29] Saeed Taheri and Ganesh Gopalakrishnan. Goat: Automated concurrency analysis and debugging tool for go. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 138–150. IEEE, 2021.

[30] Paul Thomson, Alastair F Donaldson, and Adam Betts. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 15–28, 2014.

[31] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 865–878, 2019.

[32] Uber Technologies, Inc. Goleak: Goroutine leak detector. https://github.com/uber-go/goleak, 2023.

[33] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 187–199. IEEE, 2021.

Appendix A

Detection results on GoKer

Table A.1 displays the outputs on blocking bugs, while Table A.2 displays the outputs on nonblocking bugs. These outputs are produced by our execution of the four tools on the GoKer dataset, which can be found in the Project Material attached. We have organized the outputs in a way consistent with Table 4.2, making it convenient for readers to reference. The meaning of the keywords in the tables is as follows: **X:** Not detected, **D-PA:** Partial Deadlock, **D-GL:** Global Deadlock, **MU:** Missing Unlock, **CL:** Conflict Lock, **DL:** Double Lock, **DR:** Data Race, **MC:** Misuse Channel.

Cause	SubCause	Bug	Goleak	GCatch	GFuzz	GoAT
Communication Deadlock	Channel	cockroach_2448	X	X	D-PA	X
		cockroach_24808	D-GL	X	X	D-GL
		cockroach_25456	D-GL	X	X	D-GL
		cockroach_35073	D-GL	X	X	D-GL
		cockroach_35931	D-GL	X	X	D-GL
		etcd_6857	X	D-PA	D-PA	X
		grpc_1275	D-PA	X	D-PA	D-PA
		grpc_1424	D-PA	D-PA	D-PA	D-PA
		grpc_660	D-PA	D-PA	X	D-PA
		istio_17860	D-PA	X	D-PA	D-PA
		kubernetes_38669	D-PA	X	D-PA	D-PA
		kubernetes_5316	D-PA	D-PA	X	D-PA
		kubernetes_70277	D-PA	X	X	X
		moby_21233	D-PA	D-PA	X	D-PA
		moby_33293	D-PA	D-PA	D-PA	X
		moby_4395	D-PA	D-PA	D-PA	D-PA
		syncthing_5795	D-GL	X	X	D-GL
	Condition Variable	moby_29733	D-GL	X	X	D-GL
		moby_30408	D-GL	X	X	D-GL
	Channel & Condition Variable	kubernetes_11298	D-GL	X	X	X
		moby_27782	D-PA	X	D-PA	X
	Channel & Context	cockroach_10790	X	X	D-PA	D-PA
		cockroach_13197	D-PA	X	D-PA	D-PA
		cockroach_13755	D-PA	X	X	D-PA
		cockroach_18101	D-PA	X	D-PA	D-PA
		grpc_862	D-PA	X	X	D-PA
		istio_18454	X	X	D-PA	X
		kubernetes_25331	D-PA	D-PA	D-PA	D-PA
		moby_33781	D-PA	D-PA	X	X
Mixed Deadlock	Channel & Lock	etcd_6873	X	X	X	X
		etcd_7443	X	X	X	X
		etcd_7492	D-GL	D-PA	D-PA	X
		etcd_7902	D-PA	D-PA	X	X
		grpc_1353	D-PA	X	D-PA	D-PA
		grpc_1460	D-PA	D-PA	D-PA	D-PA
		istio_16224	D-GL	X	X	D-PA
		kubernetes_10182	D-PA	X	X	X
		kubernetes_1321	X	X	X	D-PA
		kubernetes_26980	X	MU	X	D-GL
		kubernetes_6632	X	D-PA	X	X
		moby_28462	D-PA	X	D-PA	X
		serving_2137	X	X	X	X
	Channel & WaitGroup	cockroach_1055	D-GL	X	X	D-GL
		cockroach_1462	X	X	X	D-GL
	Misuse WaitGroup	moby_25384	D-PA	X	X	D-PA
Resource Deadlock	AB-BA deadlock	cockroach_10214	X	X	X	D-PA
		cockroach_7504	X	X	X	D-GL
		hugo_3251	D-GL	X	X	D-GL
		kubernetes_13135	D-PA	MU	X	X
		kubernetes_30872	X	CL	X	D-PA
		moby_4951	X	CL	X	X
	RWR deadlock	cockroach_16167	X	X	X	X
		cockroach_3710	X	DL	X	D-PA
		cockroach_6181	D-PA	DL	X	X
		kubernetes_58107	X	X	X	D-GL
		kubernetes_62464	X	DL	X	X
	Double locking	cockroach_584	D-PA	MU	X	D-PA
		cockroach_9935	D-PA	DL	X	D-PA
		etcd_5509	X	MU	X	X
		etcd_6708	D-GL	DL	X	D-GL
		etcd_10492	D-GL	DL	X	D-GL
		grpc_3017	D-GL	MU	X	D-GL
		grpc_795	D-GL	DL	X	D-GL
		hugo_5379	D-GL	DL	X	D-GL
		moby_17176	D-PA	MU	X	D-PA
		moby_36114	D-PA	DL	X	D-PA
		moby_7559	D-PA	DL	X	D-PA
		syncthing_4829	D-GL	DL	X	D-GL

Table A.1

Cause	SubCause	Bug	Goleak	GCatch	GFuzz	GoAT
Go-Specific	Anonymous function	cockroach_35501	DR	X	X	X
		kubernetes_70892	X	X	X	D-PA
		moby_22941	DR	X	X	X
		moby_27037	DR	X	X	X
	Misuse channel	etcd_3077	MC	X	X	X
		grpc_2371	MC	X	X	X
		grpc_1687	MC	X	X	X
		istio_8967	D-PA	X	X	D-PA
		serving_5865	MC	D-PA	D-PA	X
		serving_3068	MC	X	X	X
	WaitGroup	cockroach_4407	X	X	X	X
		kubernetes_13058	DR	X	X	D-PA
	Testing library	serving_6171	X	X	X	X
		serving_4908	X	X	X	X
Traditional	Data race	etcd_9446	DR	X	X	X
		etcd_8194	DR	X	X	X
		etcd_4876	DR	X	X	X
		grpc_3090	DR	X	X	X
		grpc_1748	DR	X	X	D-PA
		istio_16742	DR	X	X	X
		istio_8214	DR	X	X	X
		istio_8144	DR	X	X	X
		kubernetes_89164	DR	X	X	X
		kubernetes_88331	DR	D-PA	D-PA	D-PA
		kubernetes_82550	DR	X	X	X
		kubernetes_82239	DR	X	X	X
		kubernetes_81148	DR	X	X	X
		kubernetes_81091	X	X	X	X
		kubernetes_80284	DR	X	X	X
		kubernetes_79631	DR	D-PA	D-PA	D-PA
		kubernetes_77796	DR	X	X	X
		kubernetes_49404	DR	X	X	X
		serving_6472	X	X	X	X
		serving_3148	DR	X	X	X
	Order violation	moby_18412	DR	D-PA	D-PA	D-PA

Table A.2