Understanding Real-World Concurrency Bugs in Go

Tengfei Tu[*]
BUPT, Pennsylvania State University
tutengfei.kevin@bupt.edu.cn

Xiaoyu Liu[†]
Purdue University
liu1962@purdue.edu

Linhai Song
Pennsylvania State University
songlh@ist.psu.edu

Yiying Zhang
Purdue University
yiying@purdue.edu

Abstract

Go is a statically-typed programming language that aims to provide a simple, efficient, and safe way to build multi-threaded software. Since its creation in 2009, Go has matured and gained significant adoption in production and open-source software. Go advocates for the usage of message passing as the means of inter-thread communication and provides several new concurrency mechanisms and libraries to ease multi-threading programming. It is important to understand the implication of these new proposals and the comparison of message passing and shared memory synchronization in terms of program errors, or bugs. Unfortunately, as far as we know, there has been no study on Go's concurrency bugs.

In this paper, we perform the first systematic study on concurrency bugs in real Go programs. We studied six popular Go software including Docker, Kubernetes, and gRPC. We analyzed 171 concurrency bugs in total, with more than half of them caused by non-traditional, Go-specific problems. Apart from root causes of these bugs, we also studied their fixes, performed experiments to reproduce them, and evaluated them with two publicly-available Go bug detectors. Overall, our study provides a better understanding on Go's concurrency models and can guide future researchers and practitioners in writing better, more reliable Go software and in developing debugging and diagnosis tools for Go.

[*]The work was done when Tengfei Tu was a visiting student at Pennsylvania State University.
[†]Xiaoyu Liu contributed equally with Tengfei Tu in this work.

ASPLOS'19, April 13–17, 2019, Providence, RI, USA
© 2019 Association for Computing Machinery.
ACM ISBN ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/http://dx.doi.org/10.1145/3297858.3304069

CCS Concepts **• Computing methodologies → Concurrent programming languages**; **• Software and its engineering → Software testing and debugging**.

Keywords Go; Concurrency Bug; Bug Study

ACM Reference Format:
Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go . In *Proceedings of 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS'19).* ACM, New York, NY, USA, 14 pages. https://doi.org/http://dx.doi.org/10.1145/3297858.3304069

1 Introduction

Go [20] is a statically typed language originally developed by Google in 2009. Over the past few years, it has quickly gained attraction and is now adopted by many types of software in real production. These Go applications range from libraries [19] and high-level software [26] to cloud infrastructure software like container systems [13, 36] and key-value databases [10, 15].

A major design goal of Go is to improve traditional multi-threaded programming languages and make concurrent programming easier and less error-prone. For this purpose, Go centers its multi-threading design around two principles: 1) making threads (called *goroutines*) lightweight and easy to create and 2) using explicit messaging (called *channel*) to communicate across threads. With these design principles, Go proposes not only a set of new primitives and new libraries but also new implementation of existing semantics.

It is crucial to understand how Go's new concurrency primitives and mechanisms impact concurrency bugs, the type of bugs that is the most difficult to debug and the most widely studied [40, 43, 45, 57, 61] in traditional multi-threaded programming languages. Unfortunately, there has been no prior work in studying Go concurrency bugs. As a result, to date, it is still unclear if these concurrency mechanisms actually make Go easier to program and less error-prone to concurrency bugs than traditional languages.

In this paper, we conduct the first empirical study on Go concurrency bugs using six open-source, production-grade Go applications: Docker [13] and Kubernetes [36], two datacenter container systems, etcd [15], a distributed key-value store system, gRPC [19], an RPC library, and CockroachDB [10] and BoltDB [6], two database systems.

In total, we have studied 171 concurrency bugs in these applications. We analyzed the root causes of them, performed experiments to reproduce them, and examined their fixing patches. Finally, we tested them with two existing Go concurrency bug detectors (the only publicly available ones).

Our study focuses on a long-standing and fundamental question in concurrent programming: between message passing [27, 37] and shared memory, which of these inter-thread communication mechanisms is less error-prone [2, 11, 48]. Go is a perfect language to study this question, since it provides frameworks for both shared memory and message passing. However, it encourages the use of channels over shared memory with the belief that explicit message passing is less error-prone [1, 2, 21].

To understand Go concurrency bugs and the comparison between message passing and shared memory, we propose to categorize concurrency bugs along two orthogonal dimensions: the cause of bugs and their behavior. Along the cause dimension, we categorize bugs into those that are caused by misuse of shared memory and those caused by misuse of message passing. Along the second dimension, we separate bugs into those that involve (any number of) goroutines that cannot proceed (we call them *blocking bugs*) and those that do not involve any blocking (*non-blocking bugs*).

Surprisingly, our study shows that it is as easy to make concurrency bugs with message passing as with shared memory, sometimes even more. For example, around 58% of blocking bugs are caused by message passing. In addition to the violation of Go's channel usage rules (*e.g.*, waiting on a channel that no one sends data to or close), many concurrency bugs are caused by the mixed usage of message passing and other new semantics and new libraries in Go, which can easily be overlooked but hard to detect.

To demonstrate errors in message passing, we use a blocking bug from Kubernetes in Figure 1. The `finishReq` function creates a child goroutine using an anonymous function at line 4 to handle a request—a common practice in Go server programs. The child goroutine executes `fn()` and sends `result` back to the parent goroutine through channel `ch` at line 6. The child will block at line 6 until the parent pulls `result` from `ch` at line 9. Meanwhile, the parent will block at `select` until either when the child sends result to `ch` (line 9) or when a timeout happens (line 11). If timeout happens earlier or if Go runtime (non-deterministically) chooses the case at line 11 when both cases are valid, the parent will return from `requestReq()` at line 12, and no one else can pull `result` from `ch` any more, resulting in the child being blocked forever. The fix is to change `ch` from an unbuffered channel to a buffered one, so that the child goroutine can always send the result even when the parent has exit.

This bug demonstrates the complexity of using new features in Go and the difficulty in writing correct Go programs like this. Programmers have to have a clear understanding of goroutine creation with anonymous function, a feature

```
1     func finishReq(timeout time.Duration) r ob {
2  -    ch := make(chan ob)
3  +    ch := make(chan ob, 1)
4       go func() {
5         result := fn()
6         ch <- result // block
7       }
8       select {
9         case result = <- ch:
10          return result
11        case <- time.After(timeout):
12          return nil
13      }
14    }
```

Figure 1. A blocking bug caused by channel.

Go proposes to ease the creation of goroutines, the usage of buffered vs. unbuffered channels, the non-determinism of waiting for multiple channel operations using `select`, and the special library `time`. Although each of these features were designed to ease multi-threaded programming, in reality, it is difficult to write correct Go programs with them.

Overall, our study reveals new practices and new issues of Go concurrent programming, and it sheds light on an answer to the debate of message passing vs. shared memory accesses. Our findings improve the understanding of Go concurrency and can provide valuable guidance for future tool design. This paper makes the following key contributions.

- We performed the first empirical study of Go concurrency bugs with six real-world, production-grade Go applications.
- We made nine high-level key observations of Go concurrency bug causes, fixes, and detection. They can be useful for Go programmers' references. We further make eight insights into the implications of our study results to guide future research in the development, testing, and bug detection of Go.
- We proposed new methods to categorize concurrency bugs along two dimensions of bug causes and behaviors. This taxonomy methodology helped us to better compare different concurrency mechanisms and correlations of bug causes and fixes. We believe other bug studies can utilize similar taxonomy methods as well.

All our study results and studied commit logs can be found at *https://github.com/system-pclub/go-concurrency-bugs*.

2 Background and Applications

Go is a statically-typed programming language that is designed for concurrent programming from day one [60]. Almost all major Go revisions include improvements in its concurrency packages [23]. This section gives a brief background on Go's concurrency mechanisms, including its thread model, inter-thread communication methods, and thread synchronization mechanisms. We also introduce the six Go applications we chose for this study.

2.1 Goroutine

Go uses a concept called *goroutine* as its concurrency unit. Goroutines are lightweight user-level threads that the Go runtime library manages and maps to kernel-level threads in an M-to-N way. A goroutine can be created by simply adding the keyword go before a function call.

To make goroutines easy to create, Go also supports creating a new goroutine using an *anonymous function*, a function definition that has no identifier, or "name". All local variables declared before an anonymous function are accessible to the anonymous function, and are potentially shared between a parent goroutine and a child goroutine created using the anonymous function, causing data race (Section 6).

2.2 Synchronization with Shared Memory

Go supports traditional shared memory accesses across goroutines. It supports various traditional synchronization primitives like lock/unlock (`Mutex`), read/write lock (`RWMutex`), condition variable (`Cond`), and atomic read/write (`atomic`). Go's implementation of `RWMutex` is different from `pthread_rwlock_t` in C. Write lock requests in Go have a higher privilege than read lock requests.

As a new primitive introduced by Go, `Once` is designed to guarantee a function is only executed once. It has a `Do` method, with a function `f` as argument. When `Once.Do(f)` is invoked many times, only for the first time, `f` is executed. `Once` is widely used to ensure a shared variable only be initialized once by multiple goroutines.

Similar to `pthread_join` in C, Go uses `WaitGroup` to allow multiple goroutines to finish their shared variable accesses before a waiting goroutine. Goroutines are added to a `WaitGroup` by calling `Add`. Goroutines in a `WaitGroup` use `Done` to notify their completion, and a goroutine calls `Wait` to wait for the completion notification of all goroutines in a `WaitGroup`. Misusing `WaitGroup` can cause both blocking bugs (Section 5) and non-blocking bugs (Section 6).

2.3 Synchronization with Message Passing

Channel (`chan`) is a new concurrency primitive introduced by Go to send data and states across goroutines and to build more complex functionalities [3, 50]. Go supports two types of channels: buffered and unbuffered. Sending data to (or receiving data from) an unbuffered channel will block a goroutine, until another goroutine receives data from (or sends data to) the channel. Sending to a buffered channel will only block, when the buffer is full. There are several underlying rules in using channels and the violation of them can create concurrency bugs. For example, channel can only be used after initialization, and sending data to (or receiving data from) a `nil` channel will block a goroutine forever. Sending data to a closed channel or close an already closed channel can trigger a runtime panic.

The `select` statement allows a goroutine to wait on multiple channel operations. A `select` will block until one of its cases can make progress or when it can execute a `default` branch. When more than one cases in a `select` are valid, Go will randomly choose one to execute. This randomness can cause concurrency bugs as will be discussed in Section 6.

Go introduces several new semantics to ease the interaction across multiple goroutines. For example, to assist the programming model of serving a user request by spawning a set of goroutines that work together, Go introduces `context` to carry request-specific data or metadata across goroutines. As another example, `Pipe` is designed to stream data between a `Reader` and a `Writer`. Both `context` and `Pipe` are new forms of passing messages and misusing them can create new types of concurrency bugs (Section 5).

Application	Stars	Commits	Contributors	LOC	Dev History
Docker	48975	35149	1767	786K	4.2 Years
Kubernetes	36581	65684	1679	2297K	3.9 Years
etcd	18417	14101	436	441K	4.9 Years
CockroachDB	13461	29485	197	520k	4.2 Years
gRPC*	5594	2528	148	53K	3.3 Years
BoltDB	8530	816	98	9K	4.4 Years

Table 1. Information of selected applications. *The number of stars, commits, contributors on GitHub, total source lines of code, and development history on GitHub. *: the gRPC version that is written in Go.*

2.4 Go Applications

Recent years have seen a quick increase in popularity and adoption of the Go language. Go was the 9th most popular language on GitHub in 2017 [18]. As of the time of writing, there are 187K GitHub repositories written in Go.

In this study, we selected six representative, real-world software written in Go, including two container systems (Docker and Kubernetes), one key-value store system (etcd), two databases (CockroachDB and BoltDB), and one RPC library (gRPC-go[1]) (Table 1). These applications are open-source projects that have gained wide usages in datacenter environments. For example, Docker and Kubernetes are the top 2 most popular applications written in Go on GitHub, with 48.9K and 36.5K stars (etcd is the 10th, and the rest are ranked in top 100). Our selected applications all have at least three years of development history and are actively maintained by developers currently. All our selected applications are of middle to large sizes, with lines of code ranging from 9 thousand to more than 2 million. Among the six applications, Kubernetes and gRPC are projects originally developed by Google.

3 Go Concurrency Usage Patterns

Before studying Go concurrency bugs, it is important to first understand how real-world Go concurrent programs are like.

[1]We will use gRPC to represent the gRPC version that is written Go in the following paper, unless otherwise specified.

Application	Normal F.	Anonymous F.	Total	Per KLOC
Docker	33	112	145	0.18
Kubernetes	301	233	534	0.23
etcd	86	211	297	0.67
CockroachDB	27	125	152	0.29
gRPC-Go	14	30	44	0.83
BoltDB	2	0	2	0.22
gRPC-C	5	-	5	0.03

Table 2. Number of goroutine/thread creation sites. *The number of goroutine/thread creation sites using normal functions and anonymous functions, total number of creation sites, and creation sites per thousand lines of code.*

This section presents our static and dynamic analysis results of goroutine usages and Go concurrency primitive usages in our selected six applications.

3.1 Goroutine Usages

To understand concurrency in Go, we should first understand how goroutines are used in real-world Go programs. One of the design philoshopies in Go is to make goroutines lightweight and easy to use. Thus, we ask "do real Go programmers tend to write their code with many goroutines (*static*)?" and "do real Go applications create a lot of goroutines during runtime (*dynamic*)?"

To answer the first question, we collected the amount of goroutine creation sites (i.e., the source lines that create goroutines). Table 2 summarizes the results. Overall, the six applications use a large amount of goroutines. The average creation sites per thousand source lines range from 0.18 to 0.83. We further separate creation sites to those that use normal functions to create goroutines and those that use anonymous functions. All the applications except for Kubernetes and BoltDB use more anonymous functions.

To understand the difference between Go and traditional languages, we also analyzed another implementation of gRPC, gRPC-C, which is implemented in C/C++. gRPC-C contains 140K lines of code and is also maintained by Google's gRPC team. Compared to gRPC-Go, gRPC-C has surprisingly very few threads creation (only five creation sites and 0.03 sites per KLOC).

We further study the runtime creation of goroutines. We ran gRPC-Go and gRPC-C to process three performance benchmarks that were designed to compare the performance of multiple gRPC versions written in different programming languages [22]. These benchmarks configure gRPC with different message formats, different numbers of connections, and synchronous vs. asynchronous RPC requests. Since gRPC-C is faster than gRPC-Go [22], we ran gRPC-C and gRPC-Go to process the same amount of RPC requests, instead of the same amount of total time.

Table 3 shows the ratio of the number of goroutines created in gRPC-Go over the number of threads created in gRPC-C when running the three workloads. More goroutines are created across different workloads for both the client side and the server side. Table 3 also presents our study results of

Workload	Goroutines/Threads		Ave. Execution Time	
	client	server	client-Go	server-Go
g_sync_ping_pong	7.33	2.67	63.65%	76.97%
sync_ping_pong	7.33	4	63.23%	76.57%
qps_unconstrained	201.46	6.36	91.05%	92.73%

Table 3. Dynamic information when executing RPC benchmarks. *The ratio of goroutine number divided by thread number and the average goroutine execution time normalized by the whole application's execution time.*

Application	Shared Memory					Message		Total
	Mutex	atomic	Once	WaitGroup	Cond	chan	Misc.	
Docker	62.62%	1.06%	4.75%	1.70%	0.99%	27.87%	0.99%	1410
Kubernetes	70.34%	1.21%	6.13%	2.68%	0.96%	18.48%	0.20%	3951
etcd	45.01%	0.63%	7.18%	3.95%	0.24%	42.99%	0	2075
CockroachDB	55.90%	0.49%	3.76%	8.57%	1.48%	28.23%	1.57%	3245
gRPC-Go	61.20%	1.15%	4.20%	7.00%	1.65%	23.03%	1.78%	786
BoltDB	70.21%	2.13%	0	0	0	23.40%	4.26%	47

Table 4. Concurrency Primitive Usage. *The Mutex column includes both Mutex and RWMutex.*

goroutine runtime durations and compare them to gRPC-C's thread runtime durations. Since gRPC-Go and gRPC-C's total execution time is different and it is meaningless to compare absolute goroutine/thread duration, we report and compare the goroutine/thread duration relative to the total runtime of gRPC-Go and gRPC-C. Specifically, we calculate average execution time of all goroutines/threads and normalize it using the total execution time of the programs. We found all threads in gRPC-C execute from the beginning to the end of the whole program (i.e., 100%) and thus only included the results of gRPC-Go in Table 3. For all workloads, the normalized execution time of goroutines is shorter than threads.

Observation 1: *Goroutines are shorter but created more frequently than C (both statically and at runtime).*

3.2 Concurrency Primitive Usages

After a basic understanding of goroutine usages in real-world Go programs, we next study how goroutines communicate and synchronize in these programs. Specifically, we calculate the usages of different types of concurrency primitives in the six applications. Table 4 presents the total (absolute amount of primitive usages) and the proportion of each type of primitive over the total primitives. Shared memory synchronization operations are used more often than message passing, and `Mutex` is the most widely-used primitive across all applications. For message-passing primitives, `chan` is the one used most frequently, ranging from 18.48% to 42.99%.

We further compare the usages of concurrency primitives in gRPC-C and in gRPC-Go. gRPC-C only uses `lock`, and it is used in 746 places (5.3 primitive usages per KLOC). gRPC-Go uses eight different types of primitives in 786 places (14.8 primitive usages per KLOC). Clearly, gRPC-Go uses a larger amount of and a larger variety of concurrency primitives than gRPC-C.

Next, we study how the usages of concurrency primitives change over time. Figures 2 and 3 present the shared-memory and message-passing primitive usages in the six applications

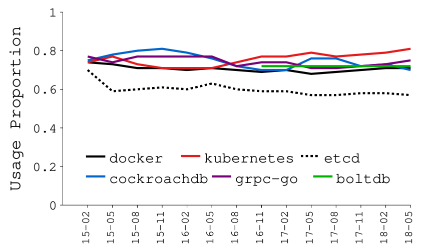

Figure 2. Usages of Shared-Memory Primitives over Time. *For each application, we calculate the proportion of shared-memory primitives over all primitives.*

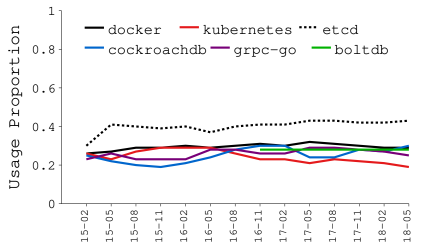

Figure 3. Usages of Message-Passing Primitives over Time. *For each application, we calculate the proportion of message-passing primitives over all primitives.*

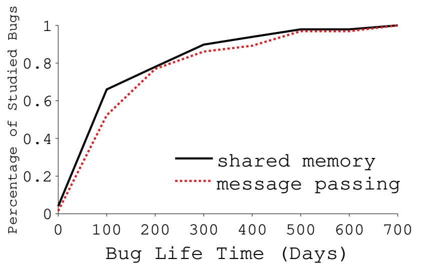

Figure 4. Bug Life Time. *The CDF of the life time of all shared-memory bugs and all message-passing bugs.*

from Feb 2015 to May 2018. Overall, the usages tend to be stable over time, which also implies that our study results will be valuable for future Go programmers.

Observation 2: *Although traditional shared memory thread communication and synchronization remains to be heavily used, Go programmers also use significant amount of message-passing primitives.*

Implication 1: *With heavier usages of goroutines and new types of concurrency primitives, Go programs may potentially introduce more concurrency bugs.*

4 Bug Study Methodology

This section discusses how we collected, categorized, and reproduced concurrency bugs in this study.

Collecting concurrency bugs. To collect concurrency bugs, we first filtered GitHub commit histories of the six applications by searching their commit logs for concurrency-related keywords, including "race", "deadlock", "synchronization", "concurrency", "lock", "mutex", "atomic", "compete", "context", "once", and "goroutine leak". Some of these keywords are used in previous works to collect concurrency bugs in other languages [40, 42, 45]. Some of them are related to new concurrency primitives or libraries introduced by Go, such as "once" and "context". One of them, "goroutine leak", is related to a special problem in Go. In total, we found 3211 distinct commits that match our search criteria.

Application	Behavior		Cause	
	blocking	non-blocking	shared memory	message passing
Docker	21	23	28	16
Kubernetes	17	17	20	14
etcd	21	16	18	19
CockroachDB	12	16	23	5
gRPC	11	12	12	11
BoltDB	3	2	4	1
Total	85	86	105	66

Table 5. Taxonomy. *This table shows how our studied bugs distribute across different categories and applications.*

We then randomly sampled the filtered commits, identified commits that fix concurrency bugs, and manually studied them. Many bug-related commit logs also mention the corresponding bug reports, and we also study these reports for our bug analysis. We studied 171 concurrency bugs in total.

Bug taxonomy. We propose a new method to categorize Go concurrency bugs according to two orthogonal dimensions. The first dimension is based on the *behavior* of bugs. If one or more goroutines are unintentionally stuck in their execution and cannot move forward, we call such concurrency issues *blocking bugs*. If instead all goroutines can finish their tasks but their behaviors are not desired, we call them *non-blocking* ones. Most previous concurrency bug studies [24, 43, 45] categorize bugs into deadlock bugs and non-deadlock bugs, where deadlocks include situations where there is a circular wait across multiple threads. Our definition of blocking is broader than deadlocks and include situations where there is no circular wait but one (or more) goroutines wait for resources that no other goroutines supply. As we will show in Section 5, quite a few Go concurrency bugs are of this kind. We believe that with new programming habits and semantics with new languages like Go, we should pay more attention to these non-deadlock blocking bugs and extend the traditional concurrency bug categorization mechanism.

The second dimension is along the *cause* of concurrency bugs. Concurrency bugs happen when multiple threads try to communicate and errors happen during such communication. Our idea is thus to categorize causes of concurrency bugs by how different goroutines communicate: by accessing shared memory or by passing messages. This categorization can help programmers and researchers choose better ways to perform inter-thread communication and to detect and avoid potential errors when performing such communication.

According to our categorization method, there are a total of 85 blocking bugs and 86 non-blocking bugs, and there are a total of 105 bugs caused by wrong shared memory protection and 66 bugs caused by wrong message passing. Table 5 shows the detailed breakdown of bug categories across each application.

We further analyzed the life time of our studied bugs, i.e., the time from when the buggy code was added (committed) to the software to when it is being fixed in the software (a bug-fixing patch is committed). As shown in Figure 4, most bugs we study (both shared memory and message passing) have long life time. We also found the time when these bugs were

Application	Shared Memory			Message Passing		
	Mutex	RWMutex	Wait	Chan	Chan w/	Lib
Docker	9	0	3	5	2	2
Kubernetes	6	2	0	3	6	0
etcd	5	0	0	10	5	1
CockroachDB	4	3	0	5	0	0
gRPC	2	0	0	6	2	1
BoltDB	2	0	0	0	1	0
Total	28	5	3	29	16	4

Table 6. Blocking Bug Causes. *Wait includes both the Wait function in* `Cond` *and in* `WaitGroup`. *Chan indicates channel operations and Chan w/ means channel operations with other operations. Lib stands for Go libraries related to message passing.*

report to be close to when they were fixed. These results show that most of the bugs we study are not easy to be triggered or detected, but once they are, they got fixed very soon. Thus, we believe these bugs are non-trivial and worth close examination.

Reproducing concurrency bugs. In order to evaluate the built-in deadlock and data-race detection techniques, we reproduced 21 blocking bugs and 20 non-blocking bugs. To reproduce a bug, we rolled the application back to the buggy version, built the buggy version, and ran the built program using the bug-triggering input described in the bug report. We leveraged the symptom mentioned in the bug report to decide whether we have successfully reproduced a bug. Due to their non-deterministic nature, concurrency bugs are difficult to reproduce. Sometimes, we needed to run a buggy program a lot of times or manually add sleep to a buggy program. For a bug that is not reproduced, it is either because we do not find some dependent libraries, or because we fail to observe the described symptom.

Threats to validity. Threats to the validity of our study could come from many aspects. We selected six representative Go applications. There are many other applications implemented in Go and they may not share the same concurrency problems. We only studied concurrency bugs that have been fixed. There could be other concurrency bugs that are rarely reproduced and are never fixed by developers. For some fixed concurrency bugs, there is too little information provided, making them hard to understand. We do not include these bugs in our study. Despite these limitations, we have made our best efforts in collecting real-world Go concurrency bugs and in conducting a comprehensive and unbiased study. We believe that our findings are general enough to motivate and guide future research on fighting Go concurrency bugs.

5 Blocking Bugs

This section presents our study results on blocking bugs, including their root causes, fixes, and the effectiveness of the built-in runtime Go deadlock detector on detecting blocking situations.

5.1 Root Causes of Blocking Bugs

Blocking bugs manifest when one or more goroutines conduct operations that wait for resources, and these resources are never available. To detect and avoid blocking bugs, it is important to understand their root causes. We study blocking bugs' root causes by examining which operation blocks a goroutine and why the operation is not unblocked by other goroutines. Using our second dimension of bug categorization, we separate blocking bugs into those that are caused by stuck operations that are intended to protect shared memory accesses and those that are caused by message passing operations. Table 6 summarizes the root causes of all the blocking bugs.

Overall, we found that there are around 42% blocking bugs caused by errors in protecting shared memory, and 58% are caused by errors in message passing. Considering that shared memory primitives are used more frequently than message passing ones (Section 3.2), message passing operations are even more likely to cause blocking bugs.

Observation 3: *Contrary to the common belief that message passing is less error-prone, more blocking bugs in our studied Go applications are caused by wrong message passing than by wrong shared memory protection.*

5.1.1 (mis)Protection of Shared Memory

Shared memory accesses are notoriously hard to program correctly and have always been one of the major focuses on deadlock research [35, 51, 54]. They continue to cause blocking bugs in Go, both with traditional patterns and new, Go-specific reasons.

Mutex 28 blocking bugs are caused by misusing locks (`Mutex`), including double locking, acquiring locks in conflicting orders, and forgetting to unlock. All bugs in this category are traditional bugs, and we believe traditional deadlock detection algorithms should be able to detect these bugs with static program analysis.

RWMutex As explained in Section 2.2, Go's write lock requests have a higher privilege than read lock requests. This unique lock implementation can lead to a blocking bug when a goroutine (th-A) acquires one *RWMutex* twice with read locking, and these two read lock operations are interleaved by a write lock operation from another goroutine (th-B). When th-A's first read lock operation succeeds, it will block th-B's write lock operation, since write locking is exclusive. However, th-B's write lock operation will also block th-A's second read lock operation, since the write lock request has a higher privilege in Go's implementation. Neither th-A nor th-B will be able to proceed.

Five blocking bugs are caused by this reason. Note that the same interleaving locking pattern will not cause blocking bugs for `pthread_rwlock_t` in C, since `pthread_rwlock_t` prioritize read lock requests under the default setting. The *RWMutex* blocking bug type implies that even when Go uses

```
1     var group sync.WaitGroup
2     group.Add(len(pm.plugins))
3     for _, p := range pm.plugins {
4       go func(p *plugin) {
5         defer group.Done()
6       }
7  -    group.Wait()
8     }
9  + group.Wait()
```

Figure 5. A blocking bug caused by `WaitGroup`.

the same concurrency semantics as traditional languages, there can still be new types of bugs because of Go's new implementation of the semantics.

Wait Three blocking bugs are due to wait operations that cannot proceed. Unlike *Mutex* and *RWMutex* related bugs, they do not involve circular wait. Two of these bugs happen when `Cond` is used to protect shared memory accesses and one goroutine calls `Cond.Wait()`, but no other goroutines call `Cond.Signal()` (or `Cond.Broadcast()`) after that.

The third bug, Docker#25384, happens with the use of a shared variable of type `WaitGroup`, as shown in Figure 5. The `Wait()` at line 7 can only be unblocked, when `Done()` at line 5 is invoked `len(pm.plugins)` times, since `len(pm.plugins)` is used as parameter to call `Add()` at line 2. However, the `Wait()` is called inside the loop, so that it blocks goroutine creation at line 4 in later iterations and it blocks the invocation of `Done()` inside each created goroutine. The fix of this bug is to move the invocation of `Wait()` out from the loop.

Although conditional variable and thread group wait are both traditional concurrency techniques, we suspect Go's new programming model to be one of the reasons why programmers made these concurrency bugs. For example, unlike `pthread_join` which is a function call that explicitly waits on the completion of (named) threads, `WaitGroup` is a variable that can be shared across goroutines and its `Wait` function implicitly waits for the `Done` function.

Observation 4: *Most blocking bugs that are caused by shared memory synchronization have the same causes and same fixes as traditional languages. However, a few of them are different from traditional languages either because of Go's new implementation of existing primitives or its new programming semantics.*

5.1.2 Misuse of Message Passing

We now discuss blocking bugs caused by errors in message passing, which in the contrary of common belief are the main type of blocking bugs in our studied applications.

Channel Mistakes in using channel to pass messages across goroutines cause 29 blocking bugs. Many of the channel-related blocking bugs are caused by the missing of a send to (or receive from) a channel or closing a channel, which will result in the blocking of a goroutine that waits to receive from (or send to) the channel. One such example is Figure 1.

```
1 - hctx, hcancel := context.WithCancel(ctx)
2 + var hctx context.Context
3 + var hcancel context.CancelFunc
4   if timeout > 0 {
5     hctx, hcancel = context.WithTimeout(ctx, timeout)
6 + } else {
7 +   hctx, hcancel = context.WithCancel(ctx)
8   }
```

Figure 6. A blocking bug caused by `context`.

When combining with the usage of Go special libraries, the channel creation and goroutine blocking may be buried inside library calls. As shown in Figure 6, a new `context` object, `hcancel`, is created at line 1. A new goroutine is created at the same time, and messages can be sent to the new goroutine through the channel field of `hcancel`. If `timeout` is larger than 0 at line 4, another `context` object is created at line 5, and `hcancel` is pointing to the new object. After that, there is no way to send messages to or close the goroutine attached to the old object. The patch is to avoid creating the extra `context` object when `timeout` is larger than 0.

```
1     func goroutine1() {
2       m.Lock()
3  -    ch <- request //blocks
4  +    select {
5  +      case ch <- request
6  +      default:
7  +    }
8       m.Unlock()
9     }
```

(a) goroutine 1

```
1  func goroutine2() {
2    for {
3      m.Lock()     //blocks
4      m.Unlock()
5      request <- ch
6    }
7  }
```

(b) goroutine 2

Figure 7. A blocking bug caused by wrong usage of channel with lock.

Channel and other blocking primitives For 16 blocking bugs, one goroutine is blocked at a channel operation, and another goroutine is blocked at lock or wait. For example, as shown in Figure 7, goroutine1 is blocked at sending `request` to channel `ch`, while goroutine2 is blocked at `m.Lock()`. The fix is to add a `select` with `default` branch for goroutine1 to make `ch` not blocking any more.

Messaging libraries Go provides several libraries to pass data or messages, like `Pipe`. These special library calls can also cause blocking bugs when not used correctly. For example, similar to channel, if a `Pipe` is not closed, a goroutine can be blocked when it tries to send data to or pull data from the unclosed `Pipe`. There are 4 collected blocking bugs caused by special Go message-passing library calls.

Observation 5: *All blocking bugs caused by message passing are related to Go's new message passing semantics like channel. They can be difficult to detect especially when message passing operations are used together with other synchronization mechanisms.*

Implication 2: *Contrary to common belief, message passing can cause more blocking bugs than shared memory. We call for attention to the potential danger in programming with message passing and raise the research question of bug detection in this area.*

	Add_s	$Move_s$	$Change_s$	$Remove_s$	Misc.
Shared Memory					
Mutex	9	7	2	8	2
Wait	0	1	0	1	1
RWMutex	0	2	0	3	0
Message Passing					
Chan	15	1	5	4	4
Chan w/	6	3	2	4	1
Messaging Lib	1	0	0	1	2
Total	31	14	9	21	10

Table 7. Fix strategies for blocking bugs. *The subscript s stands for synchronization.*

5.2 Fixes of Blocking Bugs

After understanding the causes of blocking bugs in Go, we now analyze how Go programmers fixed these bugs in the real world.

Eliminating the blocking cause of a hanging goroutine will unblock it and this is the general approach to fix blocking bugs. To achieve this goal, Go developers often adjust synchronization operations, including adding missing ones, moving or changing misplaced/misused ones, and removing extra ones. Table 7 summarizes these fixes.

Most blocking bugs caused by mistakenly protecting shared memory accesses were fixed by methods similar to traditional deadlock fixes. For example, among the 33 `Mutex`- or `RWMutex`-related bugs, 8 were fixed by adding a missing unlock; 9 were fixed by moving lock or unlock operations to proper locations; and 11 were fixed by removing an extra lock operation.

11 blocking bugs caused by wrong message passing were fixed by adding a missing message or closing operation to a channel (and on two occasions, to a pipe) on a goroutine different from the blocking one. 8 blocking bugs were fixed by adding a `select` with a `default` option (*e.g.*, Figure 7) or a case with operation on a different channel. Another common fix of channel-related blocking bugs is to replace an unbuffered channel with a buffered channel (*e.g.*, Figure 1). Other channel-related blocking bugs can be fixed by strategies such as moving a channel operation out of a critical section and replacing channel with shared variables.

To understand the relationship between the cause of a blocking bug and its fix, we apply a statistical metric called *lift*, following previous empirical studies on real-world bugs [29, 41]. *lift* is calculated as $\text{lift(A, B)} = \frac{P(AB)}{P(A)P(B)}$, where A denotes a root cause category, B denotes a fix strategy category, $P(AB)$ denotes the probability that a blocking is caused by A and fixed by B. When lift value is equal to 1, A root cause is independent with B fix strategy. When lift value is larger than 1, A and B are positively correlated, which means if a blocking is caused by A, it is more likely to be fixed by B. When *lift* is smaller than 1, A and B are negatively correlated.

Among all the bug categories that have more than 10 blocking bugs (we omit categories that have less than 10 bugs because of their statistical insignificance), *Mutex* is the category that has the strongest correlation to a type of fix—it correlates with $Move_s$ with *lift* value 1.52. The correlation between *Chan* and Add_s is the second highest, with *lift* value 1.42. All other categories that have more than 10 blocking bugs have lift values below 1.16, showing no strong correlation.

We also analyzed the fixes of blocking bugs according to the type of concurrency primitives used in the patches. As expected, most bugs whose causes are related to a certain type of primitive were also fixed by adjusting that primitive. For example, all `Mutex`-related bugs were fixed by adjusting `Mutex` primitives.

The high correlation of bug causes and the primitives and strategies used to fix them, plus the limited types of synchronization primitives in Go, suggests fruitful revenue in investigating automatic correction of blocking bugs in Go. We further find that the patch size of our studied blocking bugs is small, with an average of 6.8 lines of code. Around 90% of studied blocking bugs are fixed by adjusting synchronization primitives.

Observation 6: *Most blocking bugs in our study (both traditional shared-memory ones and message passing ones) can be fixed with simple solutions and many fixes are correlated with bug causes.*

Implication 3: *High correlation between causes and fixes in Go blocking bugs and the simplicity in their fixes suggest that it is promising to develop fully automated or semi-automated tools to fix blocking bugs in Go.*

Root Cause	# of Used Bugs	# of Detected Bugs
Mutex	7	1
Chan	8	0
Chan w/	4	1
Messaging Libraries	2	0
Total	21	2

Table 8. Benchmarks and evaluation results of the deadlock detector.

5.3 Detection of Blocking Bugs

Go provides a built-in deadlock detector that is implemented in the goroutine scheduler. The detector is always enabled during Go runtime and it reports deadlock when no goroutines in a running process can make progress. We tested all our reproduced blocking bugs with Go's built-in deadlock detector to evaluate what bugs it can find. For every tested bug, the blocking can be triggered deterministically in every run. Therefore, for each bug, we only ran it once in this experiment. Table 8 summarizes our test results.

The built-in deadlock detector can only detect two blocking bugs, BoltDB#392 and BoltDB#240, and fail in all other cases (although the detector does not report any false positives [38, 39]). There are two reasons why the built-in detector failed to detect other blocking bugs. First, it does not consider the monitored system as blocking when there are still some running goroutines. Second, it only examines whether or not goroutines are blocked at Go concurrency primitives but does not consider goroutines that wait for other systems

Application	Shared Memory				Message Passing	
	traditional	anon.	waitgroup	lib	chan	lib
Docker	9	6	0	1	6	1
Kubernetes	8	3	1	0	5	0
etcd	9	0	2	2	3	0
CockroachDB	10	1	3	2	0	0
gRPC	8	1	0	1	2	0
BoltDB	2	0	0	0	0	0
Total	46	11	6	6	16	1

Table 9. Root causes of non-blocking bugs. *traditional: traditional non-blocking bugs; anonymous function: non-blocking bugs caused by anonymous function; waitgroup: misusing WaitGroup; lib: Go library; chan: misusing channel.*

resources. These two limitations were largely due to the design goal of the built-in detector—minimal runtime overhead. When implemented in the runtime scheduler, it is very hard for a detector to effectively identify complex blocking bugs without sacrificing performance.

Implication 4: *Simple runtime deadlock detector is not effective in detecting Go blocking bugs. Future research should focus on building novel blocking bug detection techniques, for example, with a combination of static and dynamic blocking pattern detection.*

6 Non-Blocking Bugs

This section presents our study on non-blocking bugs. Similar to what we did in Section 5, we studied the root causes and fixes of non-blocking bugs and evaluated a built-in race detector of Go.

6.1 Root Causes of Non-blocking Bugs

Similar to blocking bugs, we also categorize our collected non-blocking bugs into those that were caused by failing to protect shared memory and those that have errors with message passing (Table 9).

6.1.1 Failing to Protect Shared Memory

Previous work [8, 14, 16, 17, 46, 47, 52, 62–64] found that not protecting shared memory accesses or errors in such protection are the main causes of data race and other non-deadlock bugs. Similarly, we found around 80% of our collected non-blocking bugs are due to un-protected or wrongly protected shared memory accesses. However, not all of them share the same causes as non-blocking bugs in traditional languages.

Traditional bugs More than half of our collected non-blocking bugs are caused by traditional problems that also happen in classic languages like C and Java, such as atomicity violation [8, 16, 46], order violation [17, 47, 62, 64], and data race [14, 52, 63]. This result shows that same mistakes are made by developers across different languages. It also indicates that it is promising to apply existing concurrency bug detection algorithms to look for new bugs in Go.

Interestingly, we found seven non-blocking bugs whose root causes are traditional but are largely caused by the lack of a clear understanding in new Go features. For example,

```
1     for i := 17; i <= 21; i++ { // write
2 -       go func() { /* Create a new goroutine */
3 +       go func(i int) {
4             apiVersion := fmt.Sprintf("v1.%d", i) // read
5             ...
6 -       }()
7 +       }(i)
8     }
```

Figure 8. A data race caused by anonymous function.

```
1     func (p *peer) send() {
2         p.mu.Lock()
3         defer p.mu.Unlock()
4         switch p.status {
5             case idle:
6 +               p.wg.Add(1)
7                 go func() {
8 -                   p.wg.Add(1)
9                     ...
10                    p.wg.Done()
11                }()
12            case stopped:
13        }
14    }
```

(a) func1

```
1  func (p * peer) stop() {
2      p.mu.Lock()
3      p.status = stopped
4      p.mu.Unlock()
5      p.wg.Wait()
6  }
```

(b) func2

Figure 9. A non-blocking bug caused by misusing `WaitGroup`.

Docker#22985 and CockroachDB#6111 are caused by data race on a shared variable whose reference is passed across goroutines through a channel.

Anonymous function Go designers make goroutine declaration similar to a regular function call (which does not even need to have a "function name") so as to ease the creation of goroutines. All local variables declared before a Go anonymous function are accessible by the anonymous function. Unfortunately, this ease of programming can increase the chance of data-race bugs when goroutines are created with anonymous functions, since developers may not pay enough attention to protect such shared local variables.

We found 11 bugs of this type, 9 of which are caused by a data race between a parent goroutine and a child goroutine created using an anonymous function. The other two are caused by a data race between two child goroutines. One example from Docker is shown in Figure 8. Local variable `i` is shared between the parent goroutine and the goroutines it creates at line 2. The developer intends each child goroutine uses a distinct *i* value to initialize string `apiVersion` at line 4. However, values of `apiVersion` are non-deterministic in the buggy program. For example, if the child goroutines begin after the whole loop of the parent goroutine finishes, value of `apiVersion` are all equal to 'v1.21'. The buggy program only produces desired result when each child goroutine initializes string `apiVersion` immediately after its creation and before `i` is assigned to a new value. Docker developers fixed this bug by making a copy of the shared variable `i` at every iteration and pass the copied value to the new goroutines.

Misusing WaitGroup There is an underlying rule when using `WaitGroup`, which is that `Add` has to be invoked before `Wait`. The violation of this rule causes 6 non-blocking bugs. Figure 9 shows one such bug in etcd, where there is no guarantee that Add at line 8 of func1 happens before `Wait` at line 5 of func2. The fix is to move Add into a critical section, which ensures that Add will either be executed before `Wait` or it will not be executed.

Special libraries Go provides many new libraries, some of which use objects that are *implicitly* shared by multiple goroutines. If they are not used correctly, data race may happen. For example, the `context` object type is designed to be accessed by multiple goroutines that are attached to the `context`. etcd#7816 is a data-race bug caused by multiple goroutines accessing the `string` field of a `context` object.

Another example is the `testing` package which is designed to support automated testing. A testing function (identified by beginning the function name with "Test") takes only one parameter of type `testing.T`, which is used to pass testing states such as error and log. Three data-race bugs are caused by accesses to a `testing.T` variable from the goroutine running the testing function and other goroutines created inside the testing function.

Observation 7: *About two-thirds of shared-memory non-blocking bugs are caused by traditional causes. Go's new multi-thread semantics and new libraries contribute to the rest one-third.*

Implication 5: *New programming models and new libraries that Go introduced to ease multi-thread programming can themselves be the reasons of more concurrency bugs.*

6.1.2 Errors during Message Passing

Errors during message passing can also cause non-blocking bugs and they comprise around 20% of our collected non-blocking bugs.

Misusing channel As what we discussed in Section 2, there are several rules when using channel, and violating them can lead to non-blocking bugs in addition to blocking ones. There are 16 non-blocking bugs caused by misuse of channel.

```
1 - select {
2 -   case <- c.closed:
3 -   default:
4 +     Once.Do(func() {
5         close(c.closed)
6 +     })
7 - }
```

Figure 10. A bug caused by closing a channel twice.

As an example, Docker#24007 in Figure 10 is caused by the violation of the rule that a channel can only be closed once. When multiple goroutines execute the piece of code, more than one of them can execute the `default` clause and try to close the channel at line 5, causing a runtime panic in Go. The fix is to use `Once` package to enforce that the channel is only closed once.

Another type of concurrency bugs happen when using channel and `select` together. In Go, when multiple messages received by a `select`, there is no guarantee which one will be processed first. This non-determinism implementation of `select` caused 3 bugs. Figure 11 shows one such example. The loop at line 2 executes a heavy function `f()` at line 8 whenever a ticker ticks at line 12 (case 2) and stops its execution when receiving a message from channel `stopCh` at line 10 (case 1). If receiving a message from `stopCh` and the ticker ticks at the same time, there is no guarantee which one will be chosen by `select`. If `select` chooses case 2, `f()` will be executed unnecessarily one more time. The fix is to add another `select` at the beginning of the loop to handle the unprocessed signal from `stopCh`.

```
1     ticker := time.NewTicker()
2     for {
3  +    select {
4  +      case <- stopCh:
5  +        return
6  +      default:
7  +    }
8       f()
9       select {
10        case <- stopCh:
11          return
12        case <- ticker:
13      }
14    }
```

Figure 11. A non-blocking bug caused by select and channel.

```
1  - timer := time.NewTimer(0)
2  + var timeout <- chan time.Time
3    if dur > 0 {
4  -   timer = time.NewTimer(dur)
5  +   timeout = time.NewTimer(dur).C
6    }
7    select {
8  -    case <- timer.C:
9  +    case <- timeout:
10      case <-ctx.Done():
11        return nil
12   }
```

Figure 12. A non-blocking bug caused by `Timer`.

Special libraries Some of Go's special libraries use channels in a subtle way, which can also cause non-blocking bugs. Figure 12 shows one such bug related to the `time` package which is designed for measuring time. Here, a timer is created with timeout duration 0 at line 1. At the creation time of a `Timer` object, Go runtime (implicitly) starts a library-internal goroutine which starts timer countdown. The timer is set with a timeout value `dur` at line 4. Developers here intended to return from the current function only when `dur` is larger than 0 or when `ctx.Done()`. However, when `dur` is not greater than 0, the library-internal goroutine will signal the `timer.C` channel as soon as the creation of the timer, causing the function to return prematurely (line 8). The fix is to avoid the `Timer` creation at line 1.

Observation 8: *There are much fewer non-blocking bugs caused by message passing than by shared memory accesses. Rules of channel and complexity of using channel with other*

	Timing		Instruction	Data	Misc.
	Add_s	Move_s	Bypass	Private	
Shared Memory					
traditional	27	4	5	10	0
waitgroup	3	2	1	0	0
anonymous	5	2	0	4	0
lib	1	2	1	0	2
Message Passing					
chan	6	7	3	0	0
lib	0	0	0	0	1
Total	42	17	10	14	3

Table 10. Fix strategies for non-blocking bugs. *The subscript s stands for synchronization.*

	Mutex	Channel	Atomic	WaitGroup	Cond	Misc.	None
Shared Memory							
traditional	24	3	6	0	0	0	13
waitgroup	2	0	0	4	3	0	0
anonymous	3	2	3	0	0	0	3
lib	0	2	1	1	0	1	2
Message Passing							
chan	3	11	0	2	1	2	1
lib	0	1	0	0	0	0	0
Total	32	19	10	7	4	3	19

Table 11. Synchronization primitives in patches of non-blocking bugs.

Go-specific semantics and libraries are the reasons why these non-blocking bugs happen.

Implication 6: *When used correctly, message passing can be less prone to non-blocking bugs than shared memory accesses. However, the intricate design of message passing in a language can cause these bugs to be especially hard to find when combining with other language-specific features.*

6.2 Fixes of Non-Blocking Bugs

Similar to our analysis of blocking bug fixes, we first analyze fixes of non-blocking bugs by their strategies. Table 10 categorizes the fix strategies of our studied Go non-blocking bugs, in a similar way as a previous categorization of non-blocking bug fixes in C/C++ [43].

Around 69% of the non-blocking bugs were fixed by restricting timing, either through adding synchronization primitives like `Mutex`, or through moving existing primitives like moving `Add` in Figure 9. 10 non-blocking bugs were fixed by eliminating instructions accessing shared variables or by bypassing the instructions (*e.g.*, Figure 10). 14 bugs were fixed by making a private copy of the shared variable (*e.g.*, Figure 8) and these bugs are all shared-memory ones.

To have a better understanding of non-blocking bug fixes and their relationship to bug causes, we further check what primitives are leveraged inside patches. Table 11 lists the fixes according to the type of primitives used in the patches.

Similar to the results of a previous study on patches of concurrency bugs in C/C++ [43], mutex is the most widely used primitive to enforce mutual exclusion and fix non-blocking bugs. Besides traditional bugs, mutex was also used to fix races caused by anonymous function and by `WaitGroup` and to replace misused channel.

As a new primitive, channel is the second most widely-used. Channel was leveraged to pass value between two goroutines and to replace shared variable to fix data race. It was also used to enforce the order between two operations in different goroutines. There are also bugs where channel is not properly used and is fixed in the patch (*e.g.*, Figure 10).

Interestingly, channels were not only used to fix message-passing bugs but also bugs caused by traditional shared memory synchronization. We suspect this is because some Go programmers view message passing as a more reliable way or easier-to-program way of performing inter-thread communication than shared memory synchronization.

Finally, 24 bugs were fixed by other concurrency primitives and 19 bugs were fixed without using any concurrency primitives (*e.g.*, Figure 8).

Similar to our *lift* analysis in Section 5.2, we calculate *lift* between causes and fix strategies and between causes and fix primitives for non-blocking bugs. Among bug categories with more than 10 bugs, the strongest correlation is between the cause *misusing channel* and fix primitive *channel*, with a lift value of 2.7. The cause *anonymous function* and the fix strategy *data private* has the second highest lift value of 2.23. Next, *Misusing channel* is strongly correlated with $\textit{Move}_s$ with lift value 2.21.

Observation 9: *Traditional shared memory synchronization techniques remain to be the main fixes for non-blocking bugs in Go, while channel is used widely to fix not only channel-related bugs but also shared-memory bugs.*

Implication 7: *While Go programmers continue to use traditional shared memory protection mechanisms to fix non-blocking bugs, they prefer the use of message passing as a fix in certain cases possibly because they view message passing as a safer way to communicate across threads.*

Root Cause	# of Used Bugs	# of Detected Bugs
Traditional Bugs	13	7
Anonymous Function	4	3
Lib	2	0
Misusing Channel	1	0
Total	20	10

Table 12. Benchmarks and evaluation results of the data race detector. *We consider a bug detected within 100 runs as a detected bug.*

6.3 Detection of Non-Blocking Bugs

Go provides a data race detector which uses the same happen-before algorithm as ThreadSanitizer [53]. It can be enabled by building a program using the '-race' flag. During program execution, the race detector creates up to four shadow words for every memory object to store historical accesses of the object. It compares every new access with the stored shadow word values to detect possible races.

We use our 20 reproduced non-blocking bugs to evaluate how many bugs the detector can detect. We ran each buggy program 100 times with the race detector turned on. Table 12 summarizes the number of bugs detected under each root cause category. The detector reports no false positives.

The data race detector successfully detected 7/13 traditional bugs and 3/4 bugs caused by anonymous functions. For six of these successes, the data race detector reported bugs on every run, while for the rest four, around 100 runs were needed before the detector reported a bug.

There are three possible reasons why the data race detector failed to report many non-blocking bugs. First, not all non-blocking bugs are data races; the race detector was not designed to detect these other types. Second, the effectiveness of the underlying happen-before algorithm depends on the interleaving of concurrent goroutines. Finally, with only four shadow words for each memory object, the detector cannot keep a long history and may miss data races.

Implication 8: *Simple traditional data race detector cannot effectively detect all types of Go non-blocking bugs. Future research can leverage our bug analysis to develop more informative, Go-specific non-blocking bug detectors.*

7 Discussion and Future Work

Go advocates for making thread creation easy and lightweight and for using message passing over shared memory for inter-thread communication. Indeed, we saw more goroutines created in Go programs than traditional threads and there are significant usages of Go channel and other message passing mechanisms. However, our study show that if not used correctly, these two programming practices can potentially cause concurrency bugs.

Shared memory vs. message passing. Our study found that message passing does not necessarily make multi-threaded programs less error-prone than shared memory. In fact, message passing is the main cause of blocking bugs. To make it worse, when combined with traditional synchronization primitives or with other new language features and libraries, message passing can cause blocking bugs that are very hard to detect. Message passing causes less non-blocking bugs than shared memory synchronization and surprisingly, was even used to fix bugs that are caused by wrong shared memory synchronization. We believe that message passing offers a clean form of inter-thread communication and can be useful in passing data and signals. But they are only useful if used correctly, which requires programmers to not only understand message passing mechanisms well but also other synchronization mechanisms of Go.

Implication on bug detection. Our study reveals many buggy code patterns that can be leveraged to conduct concurrency bug detection. As a preliminary effort, we built a detector targeting the non-blocking bugs caused by anonymous functions (e.g. Figure 8). Our detector has already discovered a few new bugs, one of which has been confirmed by real application developers [12].

More generally, we believe that static analysis plus previous deadlock detection algorithms will still be useful in detecting most Go blocking bugs caused by errors in shared memory synchornization. Static technologies can also help in detecting bugs that are caused by the combination of channel and locks, such as the one in Figure 7.

Misusing Go libraries can cause both blocking and non-blocking bugs. We summarized several patterns about misusing Go libraries in our study. Detectors can leverage the patterns we learned to reveal previously unknown bugs.

Our study also found the violation of rules Go enforces with its concurrency primitives is one major reason for concurrency bugs. A novel dynamic technique can try to enforce such rules and detect violation at runtime.

8 Related Works

Studying Real-World Bugs. There are many empirical studies on real-world bugs [9, 24, 25, 29, 40, 44, 45]. These studies have successfully guided the design of various bug-combating techniques. To the best of our knowledge, our work is the first study focusing on concurrency bugs in Go and the first to compare bugs caused by errors when accessing shared memory and errors when passing messages.

Combating Blocking Bugs. As a traditional problem, there are many research works fighting deadlocks in C and Java [7, 28, 33–35, 51, 54, 55, 58, 59]. Although useful, our study shows that there are many non-deadlock blocking bugs in Go, which are not the goal of these techniques. Some techniques are proposed to detect blocking bugs caused by misusing channel [38, 39, 49, 56]. However, blocking bugs can be caused by other primitives. Our study reveals many code patterns for blocking bugs that can serve the basis for future blocking bug detection techniques.

Combating Non-Blocking Bugs. Many previous research works are conducted to detect, diagnose and fix non-deadlock bugs, caused by failing to synchronize shared memory accesses [4, 5, 8, 14, 16, 17, 30–32, 43, 46, 47, 52, 62–64]. They are promising to be applied to Go concurrency bugs. However, our study finds that there is a non-negligible portion of non-blocking bugs caused by errors during message passing, and these bugs are not covered by previous works. Our study emphasizes the need of new techniques to fight errors during message passing.

9 Conclusion

As a programming language designed for concurrency, Go provides lightweight goroutines and channel-based message passing between goroutines. Facing the increasing usage of Go in various types of applications, this paper conducts the first comprehensive, empirical study on 171 real-world Go concurrency bugs from two orthogonal dimensions. Many interesting findings and implications are provided in our study. We expect our study to deepen the understanding of Go concurrency bugs and bring more attention to Go concurrency bugs.

References

[1] Principles of designing Go APIs with channels. URL: https://inconshreveable.com/07-08-2014/principles-of-designing-go-apis-with-channels/.

[2] The Go Blog: Share Memory By Communicating. URL: https://blog.golang.org/share-memory-by-communicating.

[3] Sameer Ajmani. Advanced Go Concurrency Patterns. URL: https://talks.golang.org/2013/advconc.slide.

[4] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, Texas, USA, March 2013.

[5] Joy Arulraj, Guoliang Jin, and Shan Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, Salt Lake City, Utah, USA, March 2014.

[6] boltdb. An embedded key/value database for Go. URL: https://github.com/boltdb/bolt.

[7] Yan Cai and W. K. Chan. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering, 40(3):266-281*, 2014.

[8] Lee Chew and David Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer systems (EuroSys '10)*, Paris, France, April 2010.

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM symposium on Operating Systems Principles (SOSP '01)*, Banff, Alberta, Canada, October 2001.

[10] Cockroach. CockroachDB is a cloud-native SQL database for building global, scalable cloud services that survive disasters. URL: https://github.com/cockroachdb/cockroach.

[11] Russ Cox. Bell Labs and CSP Threads. URL: http://swtch.com/ rsc/thread/.

[12] Graphql Developers. A thread-safe way of appending errors into Result.Errors. URL: https://github.com/graphql-go/graphql/pull/434.

[13] Docker. Docker - Build, Ship, and Run Any App, Anywhere. URL: https://www.docker.com/.

[14] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada, October 2010.

[15] ETCD. A distributed, reliable key-value store for the most critical data of a distributed system. URL: https://github.com/coreos/etcd.

[16] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04)*, Venice, Italy, January 2004.

[17] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2nd-strike: Toward manifesting hidden concurrency typestate bugs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, Newport Beach, California, USA, March 2011.

[18] GitHub. The fifteen most popular languages on GitHub. URL: https://octoverse.github.com/.

[19] Google. A high performance, open source, general RPC framework that puts mobile and HTTP/2 first. URL: https://github.com/grpc/grpc-go.

[20] Google. Effective Go. URL: https://golang.org/doc/effective_go.html.

[21] Google. Effective Go: Concurrency. URL: https://golang.org/doc/effective_go.html#concurrency.

[22] Google. RPC Benchmarking. URL: https://grpc.io/docs/guides/benchmarking.html.

[23] Google. The Go Programming Language – Release History. URL: https://golang.org/doc/devel/release.html.

[24] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, August 2015.

[25] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC' 14)*, Seattle, Washington, USA, November 2014.

[26] Hectane. Lightweight SMTP client written in Go. URL: https://github.com/hectane.

[27] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM, 21(8):666-677*, 1978.

[28] 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 *30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, Lincoln, Nebraska, USA, November 2015.

[29] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 12)*, Beijing, China, June 2012.

[30] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 11)*, San Jose, California, USA, June 2011.

[31] Guoliang Jin, Aditya V. Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the ACM International Conference on Object oriented programming systems languages and applications (OOPSLA '10)*, Reno/-Tahoe, Nevada, USA, October 2010.

[32] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, California, USA, October 2012.

[33] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, Dublin, Ireland, June 2009.

[34] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating systems design and implementation (OSDI '08)*, San Diego, California, USA, December 2008.

[35] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for c/pthreads. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*, Singapore, Singapore, September 2016.

[36] Kubernetes. Production-Grade Container Orchestration. URL: https://kubernetes.io/.

[37] Leslie Lamport. Concurrent Reading and Writing. *Communications of the ACM, 20(11):806-811*, 1977.

[38] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: Liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*, Paris, France, January 2017.

[39] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE '18)*, Gothenburg, Sweden, June 2018.

[40] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, Atlanta, Georgia, USA, April 2016.

[41] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability (ASID '06)*, San Jose, California, USA, October 2006.

[42] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, Lincoln, Nebraska, USA, November 2015.

[43] Haopeng Liu, Yuxi Chen, and Shan Lu. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*, Seattle, Washington, USA, November 2016.

[44] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, California, USA, February 2013.

[45] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – a comprehensive study of real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, Washington, USA, March 2008.

[46] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, San Jose, California, USA, October 2006.

[47] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09)*, New York, USA, December 2009.

[48] Kedar S. Namjoshi. Are concurrent programs that are easier to write also easier to check? In *Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[49] 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 (CC '16)*, Barcelona, Spain, March 2016.

[50] Rob Pike. Go Concurrency Patterns. URL: https://talks.golang.org/2012/concurrency.slide.

[51] Dawson R. Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03)*, Bolton Landing, New York, USA, October 2003.

[52] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems, 15(4):391-411*, 1997.

[53] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*, New York, USA, December 2009.

[54] Vivek K Shanbhag. Deadlock-detection in java-library using static-analysis. In *15th Asia-Pacific Software Engineering Conference (APSEC '08)*, Beijing, China, December 2008.

[55] Francesco Sorrentino. Picklock: A deadlock prediction approach under nested locking. In *Proceedings of the 22nd International Symposium on Model Checking Software (SPIN '15)*, Stellenbosch, South Africa, August 2015.

[56] Kai Stadtmüller, Martin Sulzmann, and Peter" Thiemann. Static trace-based deadlock analysis for synchronous mini-go. In *14th Asian Symposium on Programming Languages and Systems (APLAS '16)*, Hanoi, Vietnam, November 2016.

[57] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*, Urbana-Champaign, Illinois, USA, October 2017.

[58] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A Mahlke. Gadara: Dynamic deadlock avoidance for multi-threaded programs. In *Proceedings of the 8th USENIX Conference on Operating systems design and implementation (OSDI '08)*, San Diego, California, USA, December 2008.

[59] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '09)*, Savannah, Georgia, USA, January 2009.

[60] Wikipedia. Go (programming language). URL: https://en.wikipedia.org/wiki/Go_(programming_language).

[61] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating systems design and implementation (OSDI '10)*, Vancouver, British Columbia, Canada, October 2010.

[62] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th annual International symposium on Computer architecture (ISCA '09)*, Austin, Texas, USA, June 2009.

[63] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM symposium on Operating systems principles (SOSP '05)*, Brighton, United Kingdom, October 2005.

[64] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, Pittsburgh, Pennsylvania, USA, March 2010.