# Understanding Real-World Concurrency Bugs in Go

Lightning Talk Go Meetup Leipzig

Michael Gasch
VMware Office of the CTO

15.03.2019

## Credits

This presentation is based on the phenomenal work of

- Tengfei Tu
- Xiaoyu Liu
- Linhai Song
- Yiying Zhang

Based on their publication „Understanding Real-World Concurrency Bugs in Go“

- https://songlh.github.io/paper/go-study.pdf
- https://github.com/system-pclub/go-concurrency-bugs
- ASPLOS’19, April 13–17, 2019, Providence, RI, USA © 2019 Association for Computing Machinery

## Overview

## The Paper in a Nutshell

### From the Abstract

Go advocates for the usage of message passing as the means of inter-thread communication

It is important to understand […] the comparison of message passing and shared memory synchronization in terms of program errors, or bugs

First systematic study on concurrency bugs in real Go programs (incl. Docker, Kubernetes, gRPC)

Analyzed 171 concurrency bugs in total, with more than half of them caused by non-traditional, Go-specific problems

- Analyze root cause
- Examine fixes and patches
- Validate with Go concurrency bug detectors

## (One) Key Design Principle in Go

Improve traditional multithreaded programming languages

Make concurrent programming easier and less error-prone

Principles:

- Making threads (called goroutines) lightweight and easy to create
- Using explicit messaging (called channel) to communicate across threads

## Analysis Structure

Categorize concurrency bugs in two dimensions

- Cause of bugs by
  - Misuse of shared memory
  - Misuse of message passing
- Behavior
  - Blocking bugs
  - Non-blocking bugs

## Findings

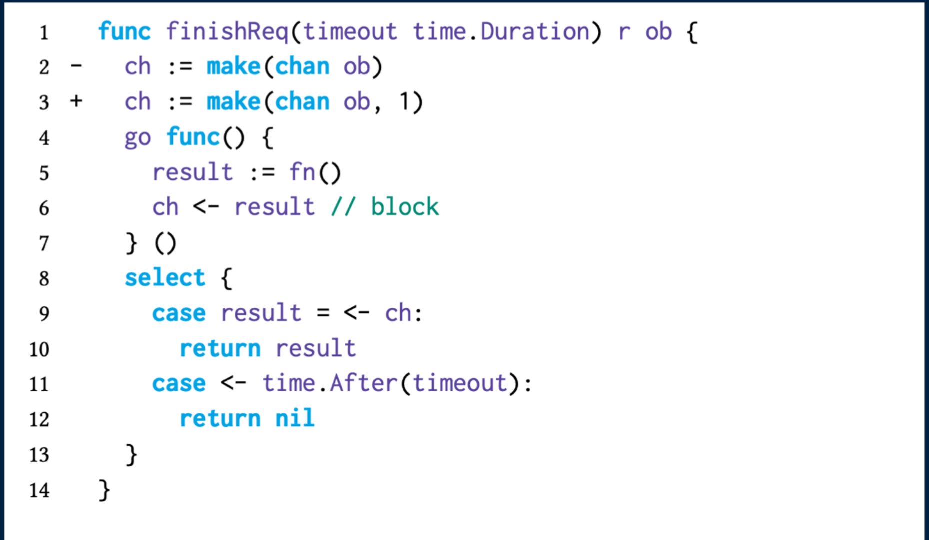
## General Findings

Easy to make concurrency bugs with message passing as with shared memory, sometimes even more

Around 58% of blocking bugs are caused by message passing

- Related: https://blogtitle.github.io/go-advanced-concurrency-patterns-part-2-timers/

Many concurrency bugs are caused by the mixed usage of message passing and other new semantics and new libraries in Go

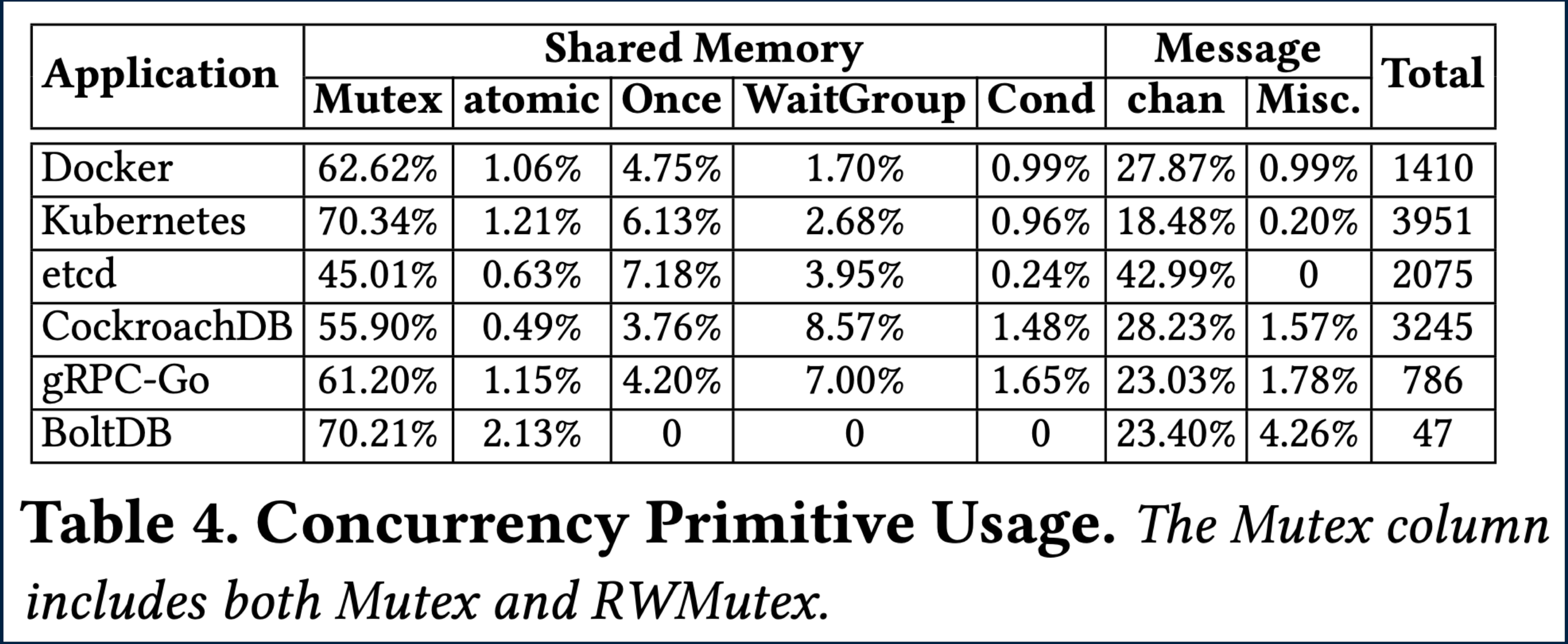


```
  func finishReq(timeout time.Duration) r ob {
-   ch := make(chan ob)
+   ch := make(chan ob, 1)
    go func() {
      result := fn()
      ch <- result // block
    } ()
    select {
      case result = <- ch:
        return result
      case <- time.After(timeout):
        return nil
    }
  }
```

**Figure 1. A blocking bug caused by channel.**

## Concurrency Primitive Usage

Mutex still preferred



| 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.*

## Bug Life Time

Majority of Concurrency Bugs is hard to detect

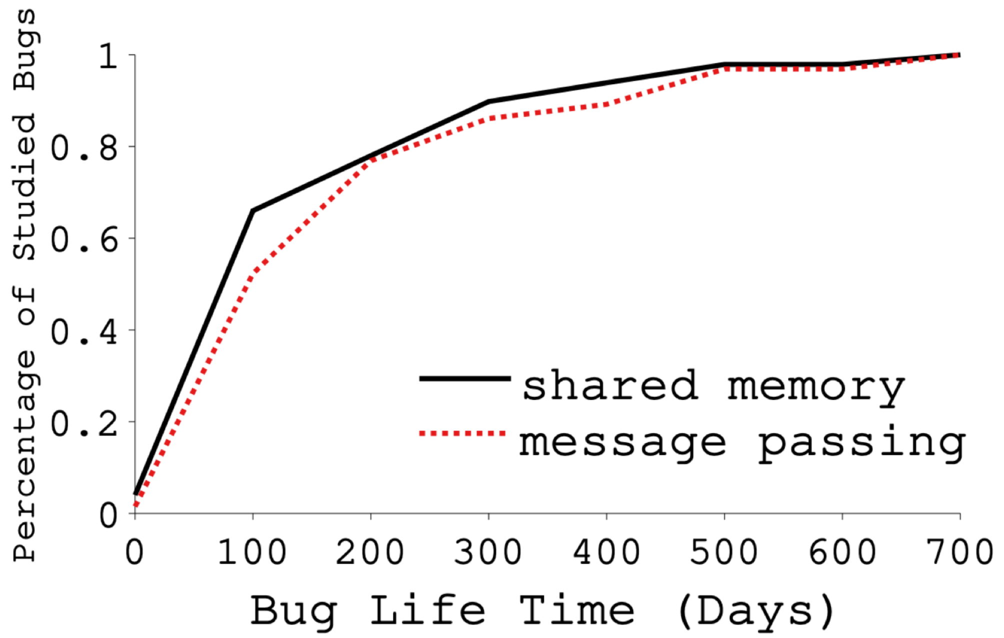
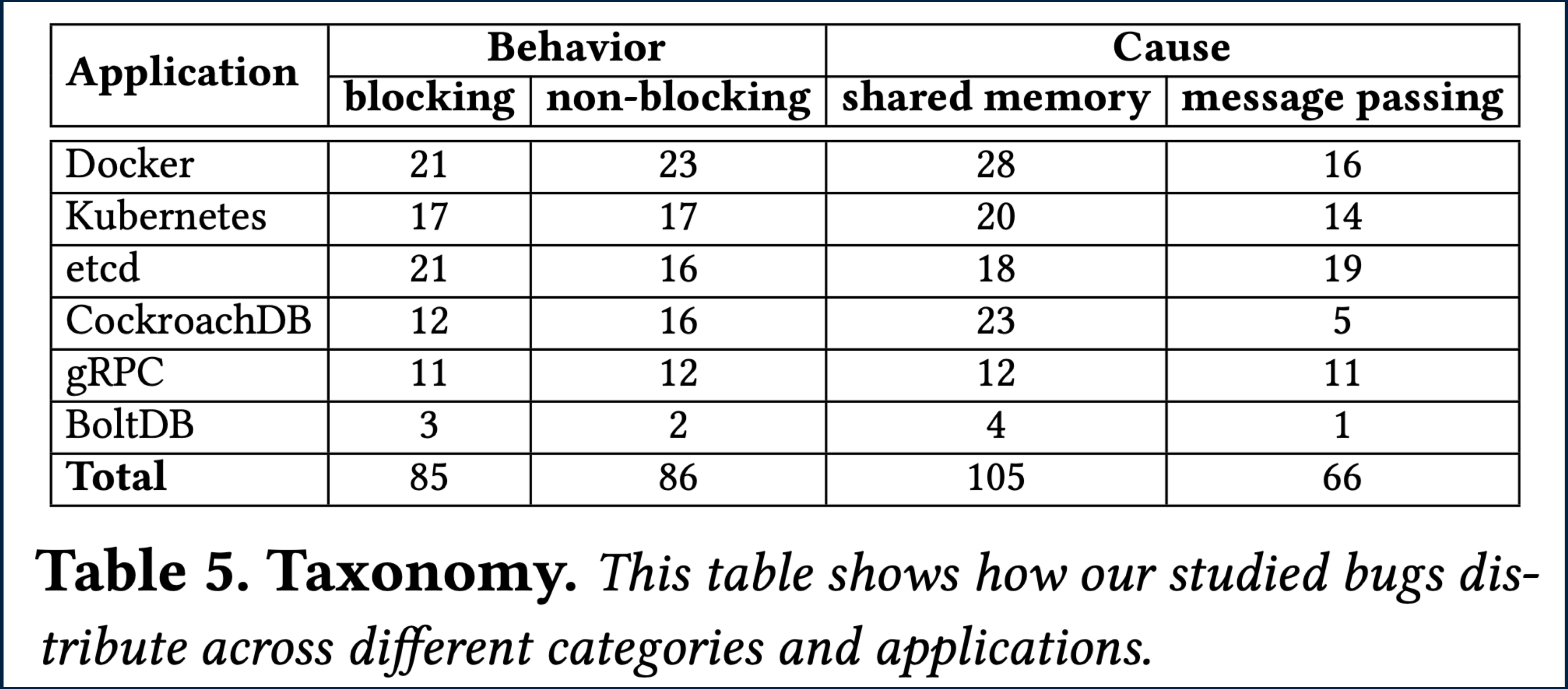


**Figure 4. Bug Life Time.** *The CDF of the life time of all shared-memory bugs and all message-passing bugs.*

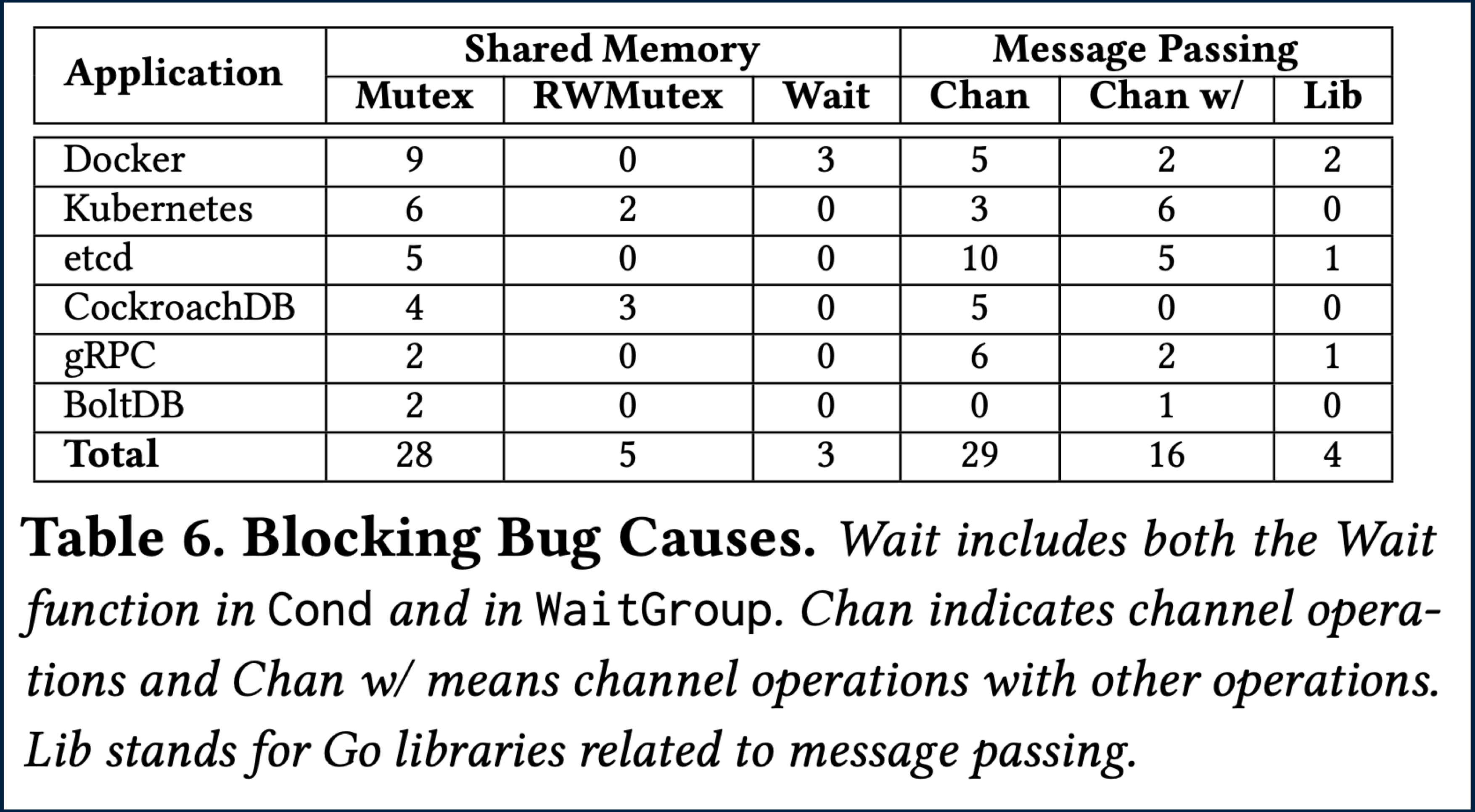
## Bug Behavior and Cause



| 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.*

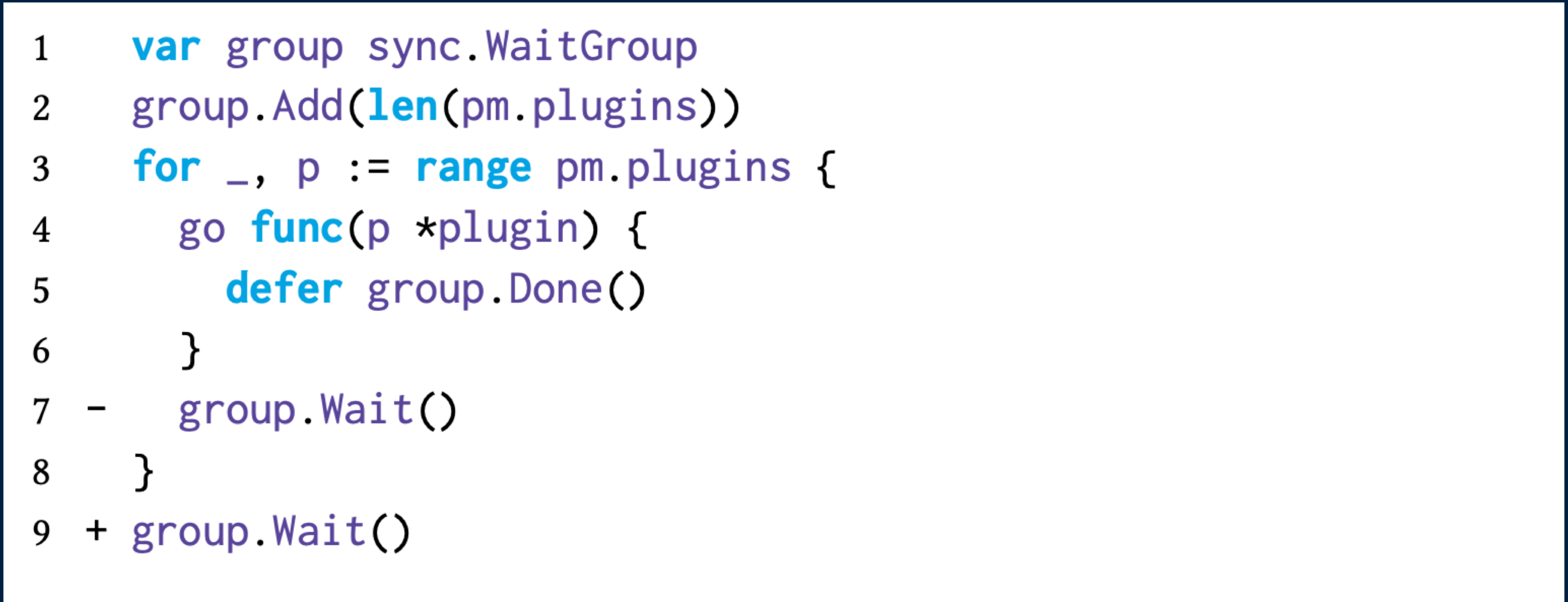
## Blocking Bug Causes



| 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.*

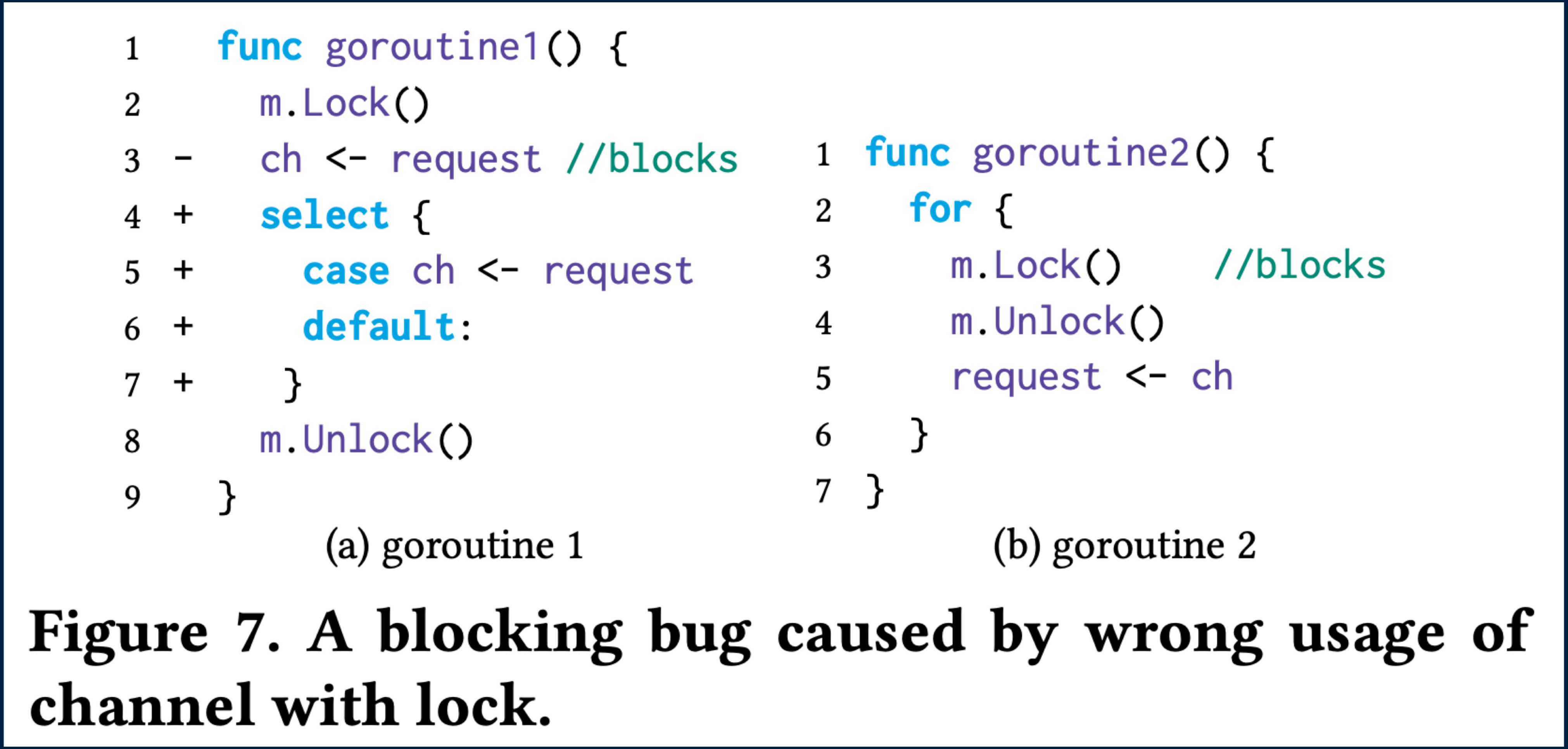
# Example



```
  var group sync.WaitGroup
  group.Add(len(pm.plugins))
  for _, p := range pm.plugins {
    go func(p *plugin) {
      defer group.Done()
    }
-   group.Wait()
  }
+ group.Wait()
```

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

# Example (2)



```
  func goroutine1() {
    m.Lock()
-   ch <- request //blocks
+   select {
+     case ch <- request
+     default:
+    }
    m.Unlock()
  }
```

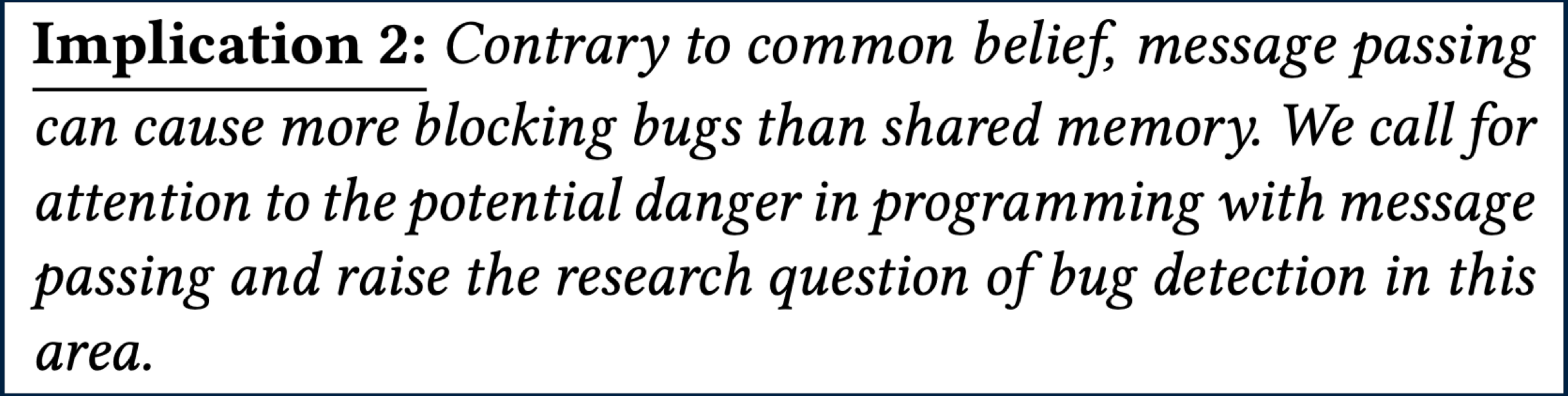
(a) goroutine 1

```
func goroutine2() {
  for {
    m.Lock()     //blocks
    m.Unlock()
    request <- ch
  }
}
```

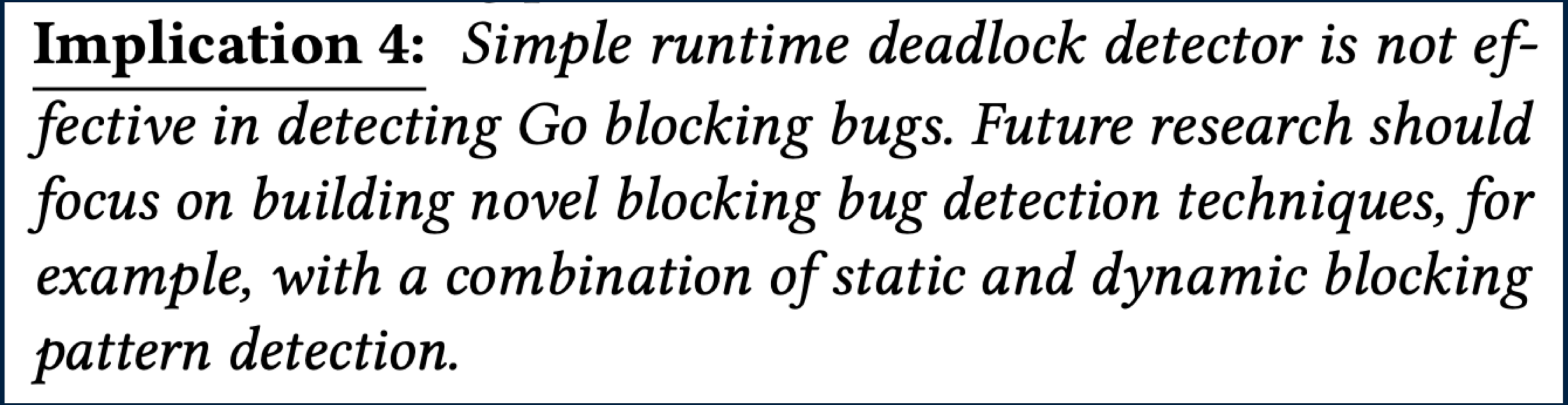
(b) goroutine 2

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

## Implications

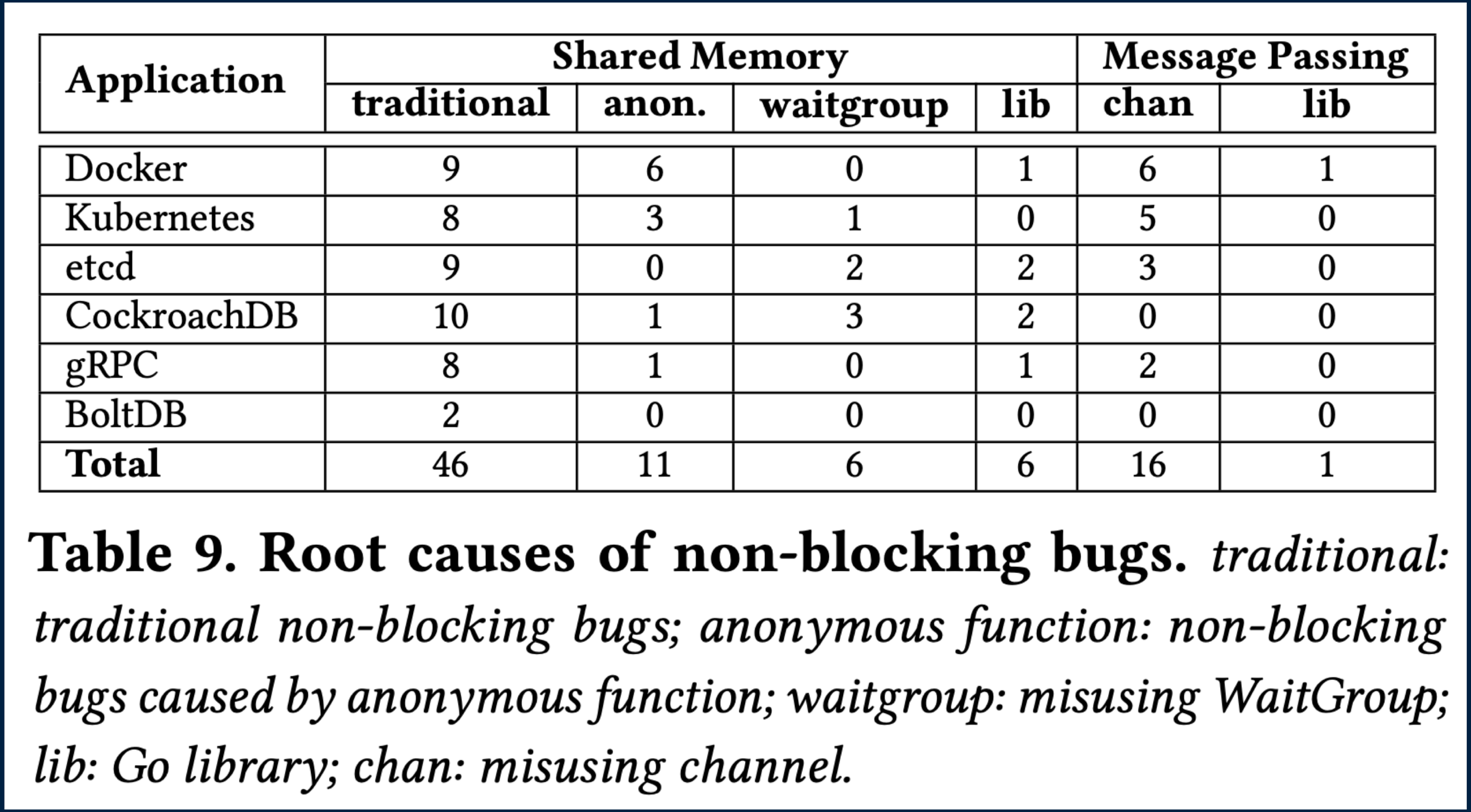


**Implication 2:** *Contrary to common belief, message passing can cause more blocking bugs than shared memory. We call for attention to the potential danger in programming with message passing and raise the research question of bug detection in this area.*



**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.*

## Non-Blocking Bug Causes

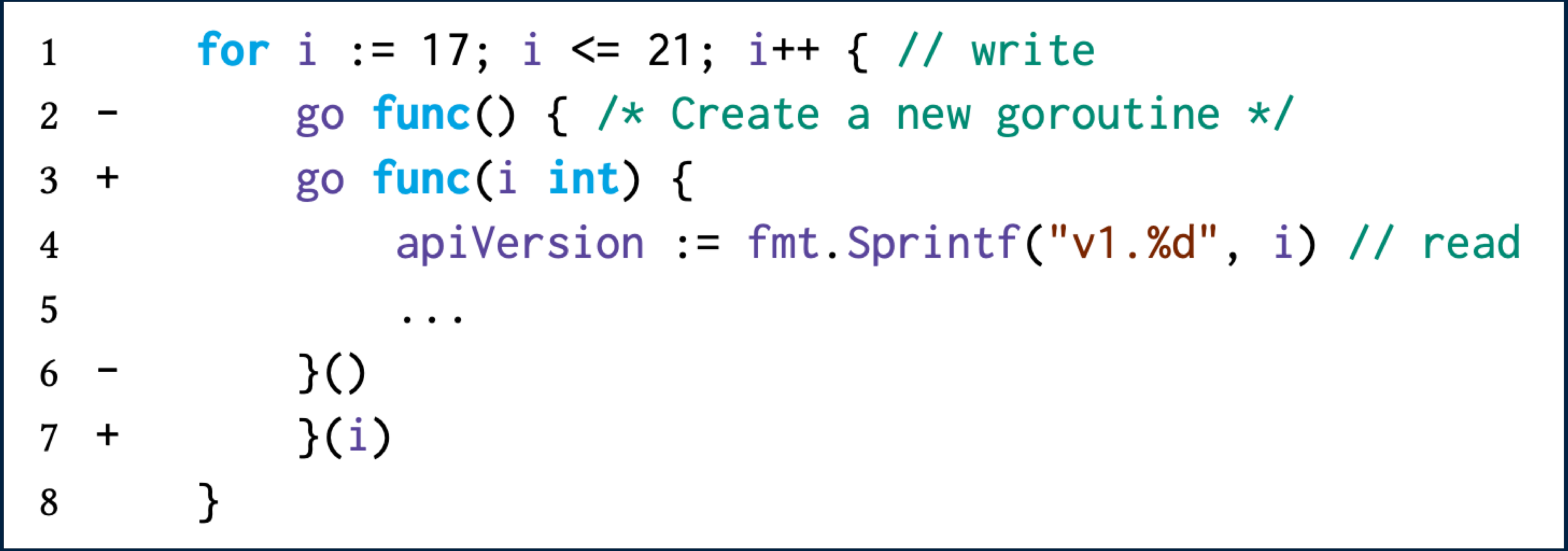


| Application | Shared Memory | | | | Message Passing | |
|---|---|---|---|---|---|---|
| | traditional | anon. | waitgroup | lib | chan | lib |
| Docker | 9 | 6 | 0 | 1 | 6 | 1 |
| Kubernetes | 8 | 3 | 1 | 0 | 5 | 0 |
| etcd | 9 | 0 | 2 | 2 | 3 | 0 |
| CockroachDB | 10 | 1 | 3 | 2 | 0 | 0 |
| gRPC | 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.*

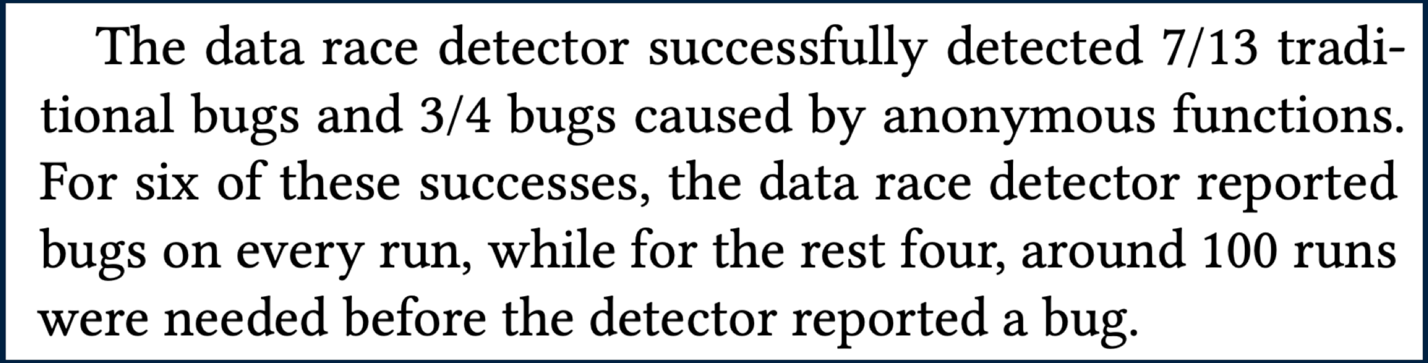
## Example

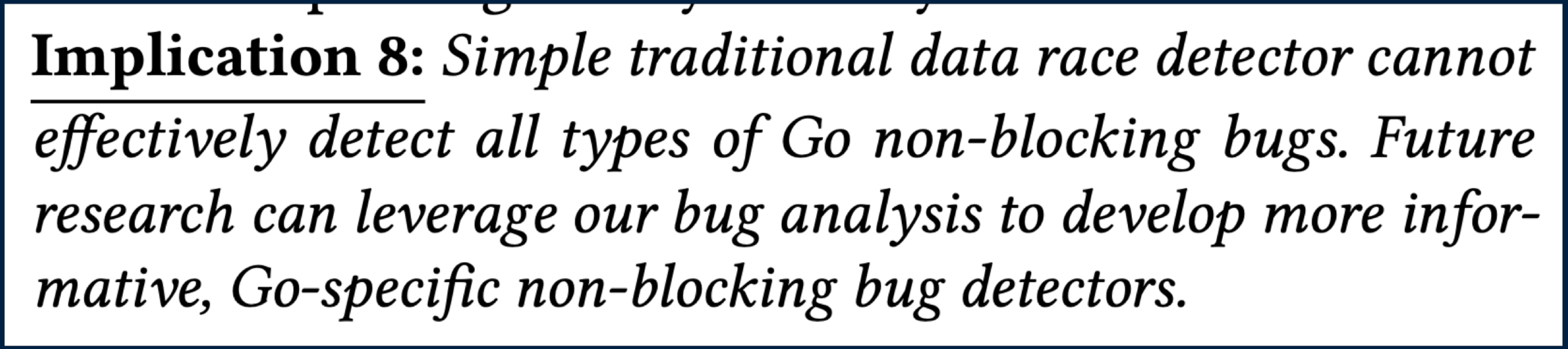
“The Classic”



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

## Implications


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.


**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.*

## Summary

## Summary

More goroutines created in Go programs than traditional threads and there are significant usages of Go channel and other message passing mechanisms

Message passing does not […] make multithreaded programs less error-prone than shared memory

Message passing is the main cause of blocking bugs

Message passing causes less nonblocking bugs than shared memory synchronization

- Was even used to fix bugs that are caused by wrong shared memory synchronization

Message passing offers a clean form of inter-thread communication and can be useful in passing data and signals

## Further Reading

# Further Reading

A static verification framework for message passing in Go using behavioural types

- https://blog.acolyer.org/2018/01/25/a-static-verification-framework-for-message-passing-in-go-using-behavioural-types/

ACIDRain: concurrency-related attacks on database backed web applications

- https://blog.acolyer.org/2017/08/07/acidrain-concurrency-related-attacks-on-database-backed-web-applications/

SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems

- https://blog.acolyer.org/2015/03/25/samc-semantic-aware-model-checking-for-fast-discovery-of-deep-bugs-in-cloud-systems/