GoBench: a Benchmark Suite of Real-World Go Concurrency Bugs

Ting Yuan[†‡], Guangwei Li[†‡], Jie Lu[†*], Chen Liu[†‡], Lian Li[†‡*] and Jingling Xue[§]
[†] State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[‡] University of Chinese Academy of Sciences, Beijing, China
[§] University of New South Wales, School of Computer Science and Engineering, Sydney, Australia
[†] {yuanting, liguangwei, lujie, liuchen17z, lianli}@ict.ac.cn [§] jingling@cse.unsw.edu.au

***Abstract*—Go, a fast growing programming language, is often considered as "the programming language of the cloud". The language provides a rich set of synchronization primitives, making it easy to write concurrent programs with great parallelism. However, the rich set of primitives also introduces many bugs.**

We build GoBench, the first benchmark suite for Go concurrency bugs. Currently, GoBench consists of 82 real bugs from 9 popular open source applications and 103 bug kernels. The bug kernels are carefully extracted and simplified from 67 out of these 82 bugs and 36 additional bugs reported in a recent study to preserve their bug-inducing complexities as much as possible. These bugs cover a variety of concurrency issues, both traditional and Go-specific. We believe GoBench will be instrumental in helping researchers understand concurrency bugs in Go and develop effective tools for their detection. We have therefore evaluated a range of representative concurrency error detection tools using GoBench. Our evaluation has revealed their limitations and provided insights for making further improvements.

I. Introduction

The Google-born Go [1] is an open-source programming language, which has gained much attention recently. It is one of the fastest growing programming languages in the software industry and is often regarded as "the programming language of the Cloud" [2]. Many popular cloud systems, such as the software container platform Docker [3] and the cluster manager Kubernetes [4], are written in Go.

Go is designed with concurrency in mind. Its designers aim to make concurrent programming easier and less error-prone by introducing two built-in concurrency constructs, i.e., goroutines and channels. Goroutines are lightweight threads managed by the Go runtime. Channels provide CSP-style message passing mechanisms [5]–[7] for communication between goroutines. Although traditional shared memory concurrency is also available, channel-based mechanisms are strongly recommended instead. It was believed that communication via message passing can make concurrent programming simpler and more reliable. However, recent surveys [8], [9] show that channel-based mechanisms introduce as much, if not more, concurrency bugs than traditional shared memory mechanisms.

Go provides a rich set of concurrency primitives. Mixing traditional locking and channel-based message passing mechanisms has introduced Go-specific bugs (e.g., communication deadlocks and channel misuse). For instance, communication deadlocks caused by channels account for more than 50% of deadlock bugs in Go applications [8], [9], but this problem was often overlooked in the past. On the other hand, the Go language and its associated tooling only provide only a basic solution for detecting concurrency bugs, including a runtime race detector and a toy global deadlock detector exist. While techniques and tools [10]–[14] have recently been proposed to detect concurrency bugs in Go, much needs to be done.

We believe that a benchmark suite can be instrumental in helping researchers understand well concurrency bugs in Go and develop effective tools for their detection. Hence, GoBench. To the best of our knowledge, this is the first benchmark suite for Go concurrency bugs. There are two test suites, a real test suite, GoReal, consisting of 82 real-world concurrency bugs, and a kernel test suite, GoKer, consisting of 103 extracted bug kernels. These bugs cover a variety of concurrency bugs, especially Go-specific bugs, such as communication deadlocks and channel misuse.

For GoReal, its 82 bugs are taken from 9 popular real-world Go applications. For each bug, we have created a Docker image to expose it on a single host machine using a `go test` command or a test script written by the original developers (for a complex bug) to ensure portability and reproducibility.

For GoKer, we have created 103 bug kernels representing 103 bugs (one bug per kernel), with 67 selected from the 82 bugs in GoReal and 36 selected from a recent study on Go concurrency bugs [9]. How and why these selections are made will be explained later in the paper. For each bug kernel, we have extracted and simplified its bug-relevant code from its corresponding source code, by preserving its bug-inducing complexities (e.g., the root cause and the bug-triggering calling sequence) as much as possible. As all the bug kernels can be compiled normally, no test scripts need to be provided.

In summary, this paper makes the following contributions:

- We introduce GoBench, the first benchmark suite to facilitate understanding and detecting concurrency bugs in Go. GoBench is publicly available at *https://github.com/timmyyuan/gobench.*
- We introduce a new taxonomy of Go concurrency bugs according to their Go-specific root causes, providing guidelines behind the development of GoBench.
- We have evaluated a range of representative tools for finding Go concurrency bugs using GoBench. Our eval-

[*] Corresponding authors.

uation has revealed their limitations and provided insights for developing improved bug-detection tools.

The rest of this paper is organized as follows. Section II reviews concurrency constructs in Go and examines different types of concurrency bugs using examples. Section III introduces GOBENCH. Section IV evaluates existing concurrency bug-detection tools using GoBench. Section V reviews the related work and Section VI concludes the paper.

II. BACKGROUND

Go is a statically typed, compiled programming language designed for concurrent programming. It offers a rich set of concurrency primitives, supporting both CSP-style concurrency and traditional shared memory mechanisms. The rich set of concurrency primitives not only facilitates concurrent programming but also bring in many concurrency bugs. In this section, we briefly introduce common concurrency mechanisms in Go and illustrate different types of concurrency bugs.

A. Goroutine

Goroutine is the concurrency unit in Go, a lightweight thread managed by the Go runtime. Unlike system threads, goroutines are cheaper to create. It is common to have hundreds of thousands of goroutines running on a single machine.

We can create a goroutine, by adding simply the keyword `go` before a function call, to run it concurrently with other functions. In practice, goroutines are often created using anonymous functions, declared in other functions. All local variables declared before an anonymous function are accessible inside. These variables are potentially shared between a parent goroutine and a child goroutine executing the anonymous function, resulting potentially in Go-specific data races.

B. Concurrency Primitives

Table I summarizes the basic synchronization primitives in Go. These can be used in shared-memory and message-passing concurrency. Those for shared-memory are packaged in the `sync` library, including mutual exclusive locks (i.e., `Mutex` and `RWMutex`), conditional variables (i.e., `Cond`), and atomic memory operations (i.e., `atomic`). These constructs are also available in languages such as Java and C/C++. `Once` is a new primitive to guarantee that a function is only executed once. When `Once.Do(foo)` is invoked multiple times, `foo` is executed only for the first invocation. `Waitgroup` is similar to thread join, where a goroutine can be used to wait for a set of goroutines in the `Waitgroup` to finish.

For message-passing, Go uses statically-typed channels to send and receive messages across the goroutines. Channels can be buffered or unbuffered. Buffered channels are asynchronous. Thus, sending (receiving) operations to a buffered channel will not block unless the buffer is full (empty). On the other hand, the sender (receiver) of an unbuffered channel will block if there is no corresponding receiver (sender) available. A channel can be set to `nil` or `closed`. Receiving messages from a closed channel will get an empty response immediately and receiving messages from a `nil` channel will block the receiver forever. Moreover, a new `switch`-like `select` statement can be used for communication across the multiple channels. A `select` statement waits for multiple channels, blocking until it can use one of its waited channels (non-deterministically) or when it can execute a `default` case.

TABLE I
CONCURRENCY PRIMITIVES IN GO.

Model	Primitive	Semantic
Shared memory	`Mutex`	a mutual exclusive lock
	`RWMutex`	a reader/writer lock
	`atomic`	an atomic memory operation
	`Cond`	a condition variable
	`Once`	exactly one action per object
	`WaitGroup`	waiting for multiple goroutines to finish
Message passing	`chan`	a channel for exchanging data between concurrent goroutines
	`select`	waiting on multiple channel operations

This rich set of concurrency primitives enables high-level concurrency patterns to be implemented in multiple ways. However, misuse of these primitives often leads to concurrency bugs, as evident in this paper and previous studies [8], [9].

C. Concurrency Bugs

Following [8], [9], [15], we classify Go concurrency bugs into blocking and non-blocking bugs. We can further differentiate bugs in each category according to their root causes.

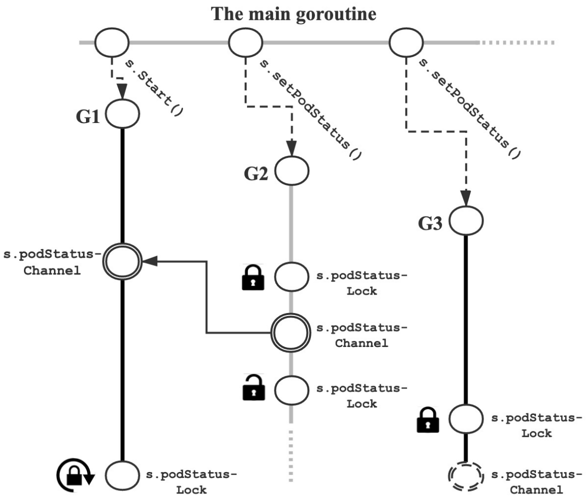
1) Blocking Bugs: These include (a) resource deadlocks (due to misuse of locks), (b) communication deadlocks (due to misuse of synchronization or channels), (c) mixed deadlocks (due to, e.g., misuse of both locks and channels). The former two are the two main sources of deadlocks [15]–[18]. A goroutine leak occurs when it waits to receive messages from a channel (post to a channel) forever, as no sender (receiver) is available. Without loss of generality, we consider goroutine leaks as a special case of blocking bugs.

a) Resource Deadlocks: A resource deadlock occurs when a set of goroutines block each other where each is waiting for resources held by some others in the set. Traditional resource deadlocks, such as double locking or acquiring locks in conflict orders (*AB-BA deadlock*), have been well studied and can be detected with existing techniques [19]–[23].

In Go, there are Go-specific resource deadlocks. For a read-write lock, `RWMutex`, a write lock request is given a higher priority than a read lock request and a goroutine can acquire the same read lock multiple times without causing double locking. Therefore, a new type of resource deadlocks can result when a goroutine (G_1) tries to acquire a write lock between the two read lock requests from another goroutine (G_2). G_1's write lock request will be blocked since G_2 has already held a read lock. However, G_2's second read lock request will also be blocked since G_1's write lock request has a higher priority. For convenience, such deadlocks are referred to as *RWR deadlocks*.

```
1       type statusManager struct {
2           podStatusesLock  sync.RWMutex
3           podStatusChannel chan bool
4       }
5       func (s *statusManager) Start() {
6           for i := 0; i < 2; i++ {
7               s.syncBatch()
8           }
9       }
10      func (s *statusManager) syncBatch() {
11          <-s.podStatusChannel
12  -       s.DeletePodStatus()
13  +       go s.DeletePodStatus()
14      }
15      func (s *statusManager) DeletePodStatus() {
16          s.podStatusesLock.Lock() // G1 blocks
17          defer s.podStatusesLock.Unlock()
18      }
19      func (s *statusManager) SetPodStatus() {
20          s.podStatusesLock.Lock()
21          defer s.podStatusesLock.Unlock()
22          s.podStatusChannel<-true // G3 blocks
23      }
24      func main() {
25          s := NewStatusManager()
26          go s.Start()          // G1 starts
27          go s.SetPodStatus()   // G2 starts
28          go s.SetPodStatus()   // G3 starts
29      }
```

a) Simplified source code



b) Steps to trigger deadlock

Fig. 1. **A mixed deadlock from `Kubernetes` (bugID `Kubernetes#10182`) [24].**

b) Communication Deadlocks: In communication deadlocks, messages are the resources for which goroutines wait. There is a set of goroutines where each is waiting for a message from another in the set. Most communication deadlocks in Go are due to misuse of channels, although traditional communication deadlocks caused by condition variables `Cond` and `WaitGroup` also exist [9]. Unlike resource deadlocks, communication deadlocks are less studied in the literature.

By default, channels are unbuffered. The send (receive) operation of an unbuffered channel in a goroutine will block until another goroutine receives data from (sends data to) the channel. If a corresponding receiver (sender) is no longer available, the sender (receiver) goroutine will block forever, resulting in so-called goroutine leaks.

c) Mixed Deadlocks: Mixed deadlocks involve channels and other concurrency primitives such as locks. They can be difficult to detect and there are no effective tools available.

Figure 1 illustrates a real deadlock bug extracted from `Kubernetes` (Table III). Figure 1(a) gives the simplified source code and Figure 1(b) depicts how the deadlock is triggered. There are 4 goroutines. The main goroutine creates the 3 goroutines G_1, G_2, and G_3 at lines 26, 27, and 28, respectively. G_1 first receives messages from `podStatusChannel` (line 11) and then tries to acquire `podStatusesLock` (line 16). G_2 and G_3 have the same entry function. Both goroutines acquire `podStatusesLock` (line 20) first and then post to `podStatusChannel` (line 22). In Figure 1(b), G_1 first receives messages from G_2 from `podStatusChannel` and then continues execution. If G_3 has successfully acquired `podStatusesLock` before G_1, then a deadlock occurs: G_1 is waiting to acquire `podStatusesLock` held by G_3, and G_3 is waiting to post to `podStatusChannel` (supposed to be received by G_1). The official fix is to create another goroutnie to acquire `podStatusesLock` (line 13).

Both communication and mixed deadlocks are common in Go applications. Therefore, and it is imperative to develop new tools and techniques targeting these problems.

```
1  -  for _, c := range checks {
2  +      for i := range checks {
3  +          c := checks[i]
4             go func() {
5                 validateCheckInTxn(&c.Name)
6             }()
7         }
8     }
```

Fig. 2. A data race from `CockroachDB#35501` [25], caused by an anonymous function.

```
1      func (s *fsSource) Stop() {
2          close(s.donec)
3  -       s.donec = nil
4      }
5      func (s *fsSource) Start() {
6          go func() {
7              select {
8                  case <-s.donec:
9                  return
10             }
11         }()
12     }
```

Fig. 3. A data race from `Istio#8967` [26], caused by channel misuse.

2) Non-blocking Bugs: Most non-blocking concurrency bugs in Go are traditional concurrency bugs caused by incorrect shared memory protection, e.g., data races and order violations. There has been a large body of research [27]–[32] on detecting such bugs. The Go compiler has implemented a classic dynamic race detector [33]. Nevertheless, it remains to be an open and challenging problem as how to effectively and efficiently detect such bugs (as evaluated in Section IV).

Some traditional concurrency bugs can occur due to Go-specific features. For instance, anonymous functions and some go libraries (e.g., `testing`) introduce implicitly shared data, leading potentially to data races. Figure 2 illustrates a data race from `CockroachDB` [34]. The race condition is highlighted in red: access to variable `c` in the parent goroutine at line 1 races with the access to `c` in the child goroutine (created by the anonymous function at line 4) at line 5. This race can be avoided by introducing a local copy of `c` at line 3.

Despite their scarcity, non-blocking bugs can occur due to misuse of channels. Figure 3 illustrates a race from `Istio` [35]. The member function `Stop` (lines 1 - 4) of of `fsSource` closes the channel `donec` and then sets it to `nil`. The other member function `Start` (lines 5 - 12) receives messages from that channel (line 8). Setting a channel to `nil` is not safe when there are concurrent communication operations on that channel. Thus, a race occurs when an instance of `fsSource` executes the two functions concurrently. To fix the bug, the developers have simply removed line 3.

III. GoBench

We introduce GoBench, the first benchmark suite for Go concurrency bugs. GoBench consists of two test suites: (1) GoReal (the real test suite) containing 82 bugs collected from 9 real-world applications, and (2) GoKer (the kernel test suite) containing 103 bugs captured by small bug kernels.

For GoReal, we have created a Docker image for each bug to allow it to be exposed on a single host machine using a `go test` command or a test script written by the original developers to ensure portability and reproducibility.

For GoKer, we have created 103 small bug kernels representing 103 bugs (one bug per kernel), with 67 selected from the 82 bugs in GoReal and 36 selected from the concurrency bugs in Go reported in a recent study [9]. We will explain this selection process below (Section III-B). For each bug kernel, we have extracted and simplified its bug-relevant code from its corresponding source code, by preserving its bug-inducing complexities (e.g., the root cause and the bug-triggering calling sequence) as much as possible. *It is non-trivial to extract such a bug kernel from a buggy application with millions of lines of code*, as explained also in Section III-B.

Table II summarizes the bugs in GoBench (i.e., GoReal and GoKer) according to a new taxonomy of Go concurrency bugs classified in terms of their Go-specific concurrency primitives (Table I). GoBench is representative, covering a diverse range of blocking and non-blocking bugs with different root causes. We have put a special emphasis on new Go-specific bugs, such as communication and mixed deadlocks. These bugs, caused by channel-based message passing, are common in Go but are not well studied in the literature.

Below, we describe how we have developed GoReal (Section III-A) and GoKer (Section III-B).

A. GoReal

We have built GoReal based on a wide range of representative concurrency bugs found in real-world applications.

We consider a set of 9 popular open-source projects listed in Table III, which cover a broad range of applications in cloud computing, from library and storage systems to containers and cluster managing systems. We have selected these projects since they are widely used in practice (e.g., `Kubernetes`, `Docker`, and `Istio`), or have a very high number of stars on GibHub (e.g., `Hugo`). We also include `Serving`, a well-known project in a new area of cloud computing.

TABLE II
Bugs in GoBench (with the number of bugs of each type given).

Suite	Bug Type (**#Bugs**)		
GoReal	Blocking (**40**)	Resource Deadlock (**9**)	Double Locking (**7**)
			AB-BA Deadlock (**2**)
		Communication Deadlock (**21**)	Channel (**16**)
			Condition Variable (**2**)
			Channel & Context (**2**)
			Channel & Condition Variable (**1**)
		Mixed Deadlock (**10**)	Channel & Lock (**8**)
			Channels & WaitGroup (**2**)
	Non-blocking (**42**)	Traditional (**24**)	Data race (**22**)
			Order Violation (**2**)
		Go-specific (**18**)	Anonymous Function (**4**)
			Channel Misuse (**6**)
			Special Libraries (**8**)
	Total		**82**
GoKer	Blocking (**68**)	Resource Deadlock (**23**)	Double Locking (**12**)
			AB-BA Deadlock (**6**)
			RWR Deadlock (**5**)
		Communication Deadlock (**29**)	Channel (**17**)
			Condition Variable (**2**)
			Channel & Context (**8**)
			Channel & Condition Variable (**2**)
		Mixed Deadlock (**16**)	Channel & Lock (**13**)
			Channel & WaitGroup (**2**)
			Misuse WaitGroup (**1**)
	Non-blocking (**35**)	Traditional (**21**)	Data race (**20**)
			Order Violation (**1**)
		Go-specific (**14**)	Anonymous Function (**4**)
			Channel Misuse (**6**)
			Special Libraries (**4**)
	Total		**103**

TABLE III
Nine studied projects.

Project	#Lines (KLOC)	#Bugs (GoReal/GoKer)	Description
`Kubernetes` [4]	3,340	21/25	Container manager
`Docker` [3]	1,067	5/16	Container framework
`Hugo` [36]	99	2/2	Static site generator
`Syncthing` [37]	80	2/2	File synchronization system
`Serving` [38]	1,171	11/7	Serverless computing
`Istio` [35]	222	7/7	Service mesh
`CockroachDB` [34]	1,594	13/20	Distributed SQL database
`Etcd` [39]	533	10/12	Distributed key-value store
`Grpc-go` [40]	98	11/12	RPC library

To collect their concurrency bugs, we search all the pull requests in the GitHub repositories of these projects using the following keywords: “deadlock”, “goroutine leak”, and “blocking” for blocking bugs, and “race”, “atomic”, and “concurrently” for non-blocking bugs. In GitHub, a pull request is a request for merging one branch into another. Developers often submit a pull request after having fixed an issue. Our keyword-based search has generated a total of 5,100 pull requests, with 1,074 on blocking issues and 3,026 on non-blocking issues. We have inspected all the 1,074 blocking-related pull requests and 1108 non-blocking-related pull requests in the last two years. In addition, we have taken advantage of 152 pull requests transformed from the commits published in [9] for the five common projects considered, `Kubernetes`, `Docker`, `CockroachDB`, `Etcd`, and `Grpc-go`. Finally, a pull request is selected for further investigation by us if (1) it addresses a real concurrency bug (as it was merged into the main trunk),

(2) there is a detailed description on how to reproduce the bug, and (3) there is a *test function* as the entry point (i.e., effectively the main function) to expose the bug. In the end, a total of 386 pull requests have been selected.

For each selected pull request related to a project, we then try to reproduce its corresponding bug. To this end, we roll back the project to the buggy version, i.e., the version before the pull request was merged. We extract the bug-triggering test function based on the pull request and run it in both the buggy application and the fixed version. A bug is considered to be reproduced if the test function fails in the buggy version but succeeds in the fixed version. For a blocking bug, its bug-triggering test function usually checks for its execution time. In this case, the test function fails if it cannot run to completion in a given period of time. For a non-blocking bug, we rely on Go's built-in dynamic race checker if it is race-related and observe the symptoms described in the pull request (e.g., a runtime panic) otherwise. We also further verify the failed traces to confirm that the bug has been triggered.

We have succeeded in reproducing 82 bugs out of 386 pull requests. For the remaining 304 bugs that we have failed to reproduce, there are various reasons behind:

- 180 bugs: We failed to reproduce these bugs after tens of thousands of runs according to the given instructions.
- 78 bugs: These bugs are system-specific, often involving too many system-specific code modifications.
- 27 bugs: These bugs can only be triggered under specific running environments, such as a cluster or a continuous integration system when running stress testing.
- 12 bugs: These bugs do not have their associated test functions. For each bug, the test function submitted by the developers is used to verify the correctness of newly added code instead of triggering the reported bug. When the developers have submitted the corresponding test case function later, the code repository has changed a lot since.
- 6 bugs: These bugs depend on some third-party libraries, which are no longer available.
- 1 bug: This is not a bug for the latest version of Go.

Finally, for each bug, we create a Docker image to allow it to be reproduced on a single host machine using a `go test` command or a test script written by the original developers.

B. GoKer

GoKer consists of 103 bug kernels (one bug per kernel), with 67 bugs taken from GoReal and 36 bugs taken from a recent study on real-world Go concurrency bugs [9]. Their code sizes range from 17 LOC to 246 LOC, with an average of 72 LOC. Unlike the bugs in GoReal, the bugs in GoKer take much less time to expose. Thus, GoKer is expected to facilitate understanding concurrency bugs in Go and developing effective tools for their detection.

When building GoKer, we have ignored a total of 15 bugs in GoReal, since they (1) rely on third-party libraries as in `Grpc-go`, (2) result in duplicated bug kernels, (3) use more than 10 goroutines, or (4) have complex interactions with other goroutines involving, for example, `gRPC` and reflection.

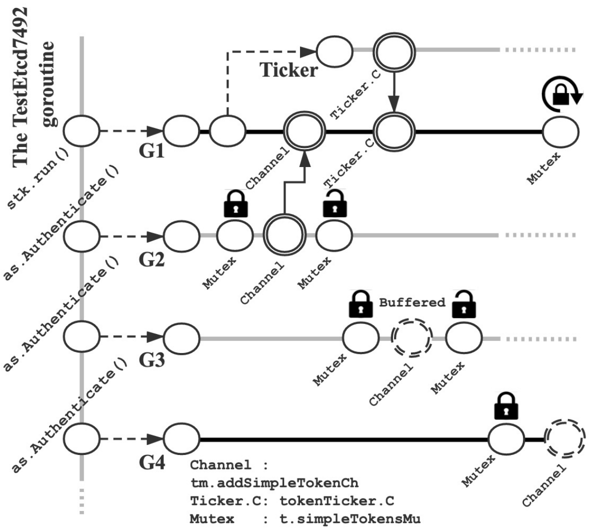


Fig. 4. Steps required for triggering `etcd#7492`.

We describe below how to create a bug kernel from a bug. It is important to emphasize that it is non-trivial to extract such a bug kernel from a buggy application with millions of lines of code, especially we would like to the bug kernel to preserve the same level of bug-inducing complexities in the original bug application (in terms of the root cause and calling sequence).

We now describe how to build a bug kernel by an example.

1) Bug `etcd#7492`: We quote the discussion (from the original developers) regarding the issue below:

> There's a deadlock in `assignSimpleTokenToUser`. The function acquires lock `as.simpleTokensMu` and posts to `addSimpleTokenCh` (suppose that the channel is full so it blocks). If the goroutine `simpleTokenTTL-Keeper.run` happens to hit `<-tokenTicker.C`, it will try to acquire `simpleTokensMu` while calling `delete-TokenFunc`. Since only the goroutine `simpleToken-TTLKeeper.run` can drain `addSimpleTokenCh`, the lock is never released.

Figure 4 depicts how the bug is triggered. There are 5 goroutines (G_1, G_2, G_3, G_4, and the main goroutine), 1 mutex (`t.simpleTokensMu`), and 2 channels (`tm.addSimpleTokenCh` and `tokenTicker.c`). Here, `tm.addSimpleTokenCh` is a bufferedc channel of size 1, and `tokenTicker.c` is for receiving ticker message from the system at regular intervals. G_1 waits for messages from `tm.addSimpleTokenCh` and `tokenTicker.C`. When a ticker message has arrived, G_1 tries to acquire the mutex `t.simpleTokensMu`. G_2, G_3 and G_4 all acquire this same mutex and send messages to `tm.addSimpleTokenCh`. A deadlock results if G_4 acquires the mutex before G_1 and the buffered channel `tm.addSimpleTokenCh` is full.

2) Abstract Data Structure: As shown in Figure 5, the channel `tm.addSimpleTokenCh` and mutex `t.simpleTokensMu` are defined in type `tokenSimple` (lines 4 - 7) and `simpleTokenTTLKeeper` (lines 14 - 22), respectively. Go uses composition and interfaces to achieve code reuse and polymorphism. `simpleTokenTTlKeeper` is composed in `tokenSimple`

```
 1      type TokenProvider interface {
 2          assign()
 3      }
 4      type tokenSimple struct {
 5          simpleTokenKeeper *simpleTokenTTLKeeper
 6          simpleTokensMu    sync.RWMutex
 7      }
 8      func (t *tokenSimple) assign() {
 9          t.assignSimpleTokenToUser()
10      }
11      type authStore struct {
12          tokenProvider TokenProvider
13      }
14      type simpleTokenTTLKeeper struct {
15          tokens              map[string]time.Time
16  -       addSimpleTokenCh    chan string
17  +       addSimpleTokenCh    chan struct{}
18  -       resetSimpleTokenCh  chan string
19  -       deleteSimpleTokenCh chan string
20  -       stopCh              chan chan struct{}
21          deleteTokenFunc     func(string)
22      }
```

Fig. 5. Abstracted types in etcd#7492.

```
G1 [semacquire]:
.../auth.newDeleter.func1(...)
.../auth.(*simpleTokenTTLKeeper).run(...)
created by .../etcd/auth.NewSimpleTokenTTLKeeper

G4 [chan send]:
.../auth.(*simpleTokenTTLKeeper).addSimpleToken(...)
.../auth.(*tokenSimple).assignSimpleTokenToUser(...)
.../auth.(*tokenSimple).assign(...)
.../auth.(*authStore).Authenticate(...)
created by .../etcd/auth.TestHammerSimpleAuthenticate
```

Fig. 6. Dumped stack traces of G_1 and G_4 (simplified).

```
23  func (tm *simpleTokenTTLKeeper) run() {
24      tokenTicker := time.NewTicker(time.Nanosecond)
25      defer tokenTicker.Stop()
26      for {
27          select {
28              case <-tm.addSimpleTokenCh:
29                  m.tokens["1"] = time.Now()
30              case <-tokenTicker.C:
31                  for t, _ := range tm.tokens {
32                      tm.deleteTokenFunc(t)
33                      delete(tm.tokens, t)
34                  }
35                  ... ...
36          }
37      }
38  }
39  func newDeleter(t *tokenSimple) func(string) {
40      return func(tk string) {
41          t.simpleTokensMu.Lock()
42          defer t.simpleTokensMu.Unlock()
43      }
44  }
45  func NewSimpleTokenTTLKeeper(deletefunc func(string))
46      *simpleTokenTTLKeeper {
47      stk := &simpleTokenTTLKeeper{
48          tokens:              make(map[string]time.Time),
49          addSimpleTokenCh:    make(chan bool, 1),
50          deleteTokenFunc:     deletefunc,
51      }
52      go stk.run() // G1
53      return stk
54  }
55  func setupAuthStore() *authStore {
56      t := &tokenSimple{}
57      t.simpleTokenKeeper = NewSimpleTokenTTLKeeper(
58          newDeleter(t))
59      return &authStore{tokenProvider: t}
60  }
```

Fig. 7. Simplified code snippet for forking G_1 (with run as its entry and setupAuthStore and NewSimpleTokenTTLKeeper as its callers)

```
61  func (tm *simpleTokenTTLKeeper) addSimpleToken() {
62      tm.addSimpleTokenCh <- struct{}{}
63  }
64  func (t *tokenSimple) assignSimpleTokenToUser() {
65      t.simpleTokensMu.Lock()
66      t.simpleTokenKeeper.addSimpleToken()
67      t.simpleTokensMu.Unlock()
68  }
69  func (as *authStore) Authenticate() {
70      as.tokenProvider.assign()
71  }
```

Fig. 8. Simplified code snippet of G_2, G_3, and G_4 (forked via the same anonymous function, which invokes Autheticate).

(line 5). Structure tokenSimple, which implements the interface TokenProvider (lines 8 - 10), is composed in structure authStore (line 12). We will preserve such composition and interfaces in the extracted bug kernel.

However, we will skip structure members irrelevant to the underlying bug. For simpleTokenTTLKeeeper, the 3 members defined in lines 18 - 20 are omitted. For a typed channel, we replace its original type with an empty structure (i.e., string with struct{} here) to avoid unnecessary dependencies (lines 16 and 17). Such abstraction is applied to tokenSimple and the other types related to the bug.

3) Preserving Call Traces: We will try to preserve the same bug-triggering traces in the extracted bug kernel. When a program asserts or timeouts, the Go runtime will dump the call traces of the running goroutines, starting from the points where they are forked. In our example, Figure 6 gives the dumped stack traces of the two blocking goroutines. G_1 is created in function NewSimpleTokenTTLKeeper, which is called in function setupAuthStore. The mutex under consideration is acquired in function newDeleterFunc.func1, which is invoked in simpleTokenTTLKeeper.run.

Figure 7 gives the simplified code for forking G_1. In function setupAuthStore (lines 55 - 60), object t of structure tokenSimple is created at line 56. At line 57, an object of structure NewSimpleTokenTTLKeeper is created and composed in t. The anonymous function (lines 40 - 43) returned from newDeleter is passed as an argument to the constructor. Note that the function will acquire the mutex t.simpleTokensMu. In Go, functions as first-class citizens are frequently assigned to variables or passed as arguments. Such dependencies are preserved in the extracted kernel.

In the constructor (lines 45 - 54), the member deleteTokenFunc is initialized to the input function (line 50), then it forks the goroutine G_1 at line 52. In G_1's entry function run (lines 23 - 38), the goroutine waits for messages from multiple channels via the select statement (lines 27 - 36). When a ticker message from tokenTicker.C arrives, G_1 invokes its member function deleteTokenFunc (line 32), i.e., the anonymous function returned from newDeleter. G_1 is then blocked at line 41, trying to acquire the mutex t.simpleTokensMu.

Similarly, we extract the simplified code snippet of G_2 (identical as G_3 and G_4) as shown in

```
72      func TestEtcd7492(t *testing.T) {
73          ... ...
74          as := setupAuthStore()    // Fork G1
75          var wg sync.WaitGroup
76  -       wg.Add(len(users))
77  +       wg.Add(3)
78  -       for u := range users {
79  +       for i := 0;i < 3; i ++ {
80  -           go func(user string) {
81  +           go func() {    // Fork G2, G3, and G4
82                  defer wg.Done()
83  -               ... ...
84  -               _, err := as.AuthInfoFromCtx(ctx)
85  -               if err != nil {
86  -                   t.Fatal(err)
87  -               }
88  -           }(u)
89  +               as.Authenticate()
90  +           }()
91          }
92  -       time.Sleep(time.Millisecond)
93          wg.Wait()
94          ... ...
95      }
```

Fig. 9. Simplified code snippet of the main goroutine.

Figure 8. The entry function `Authenticate` invokes `tokenSimple.assignSimpleTokenToUser` (lines 64 - 68) at line 70. The callee function acquires a lock (line 65) first, and then invokes `addSimpleToken` (line 66) where a message is posted to `tm.addSimpleTokenCh` (line 62). We try to preserve the inter-procedural control flows of the application in the extracted bug kernel. As such, the bug-triggering call traces as shown in Figure 6 are preserved.

4) Simplifying Control Flows: In the bug kernel, only the statements directly related to the bug are included (and all irrelevant statements are skipped). Figure 9 gives the simplified code snippet for the main goroutine. The function `setupAuthStore` is invoked at line 74, which will fork G_1 in a callee function. We preserve the `waitGroup` primitive and change the number of goroutines in the wait group to 3, the minimum needed to trigger the bug (lines 76 - 77). The number of iterations of the `for` loop (line 78 - 79) is modified accordingly. In each loop iteration, a goroutine is forked via the anonymous function (lines 80 - 81), with its body containing now a direct call to `Authenticate` (line 89). The loop will fork G_2, G_3, and G_4 to trigger the error.

Despite its code reduction, a bug kernel is extracted and simplified to preserve (as much as possible) the level of complexities of the bug-relevant code in the original application. To detect the bug in this bug kernel, we will still need to address the challenges faced in reasoning about object composition, first-class functions, indirect function calls, buffered asynchronous channels, and thread interleavings.

IV. Evaluation

To demonstrate the benefits of GoBench, we have evaluated a number of representative open-source tools for detecting concurrency bugs in GoBench. Our evaluation has revealed their limitations and provided insights for improving these tools. We have considered the following four tools:

- *goleak [11]*: This goroutine leak detector from Uber declares that a deadlock has occurred if the main goroutine fails to finish within a pre-defined time period, and reports the remaining user-defined goroutines as being deadlocked. *leaktest [12]*, which is embedded in `cockroachDB`, is similar and thus omitted.
- *go-deadlock [10]*: This tool applies traditional techniques to detect lock-related deadlocks, e.g., double locking and AB-BA deadlocks. It also reports a deadlock if acquiring a lock takes too long (30 seconds by default).
- *dingo-hunter [13], [14]*: This is a static verifier targeting only communication deadlocks caused by channels.
- *Go-runtime race detector* (*Go-rd*): The Go-runtime provides a built-in thread sanitizer to detect races at runtime.

Currently, GoBench does not have any global deadlock bug. For global deadlocks, the Go-runtime provides a built-in global deadlock detector, *Go-runtime deadlock detector*, which declares that a global deadlock has occurred when all the running goroutines in the program cannot progress anymore.

Of the four bug-detection tools evaluated, *goleak*, *go-deadlock* and *Go-rd* are dynamic while *dingo-hunter* is static. For blocking bugs, *goleak*, *go-deadlock* and *dingo-hunter* come into play. For non-locking bugs, *Go-rd* is the only open-source tool that we are aware of for detecting data races.

We have used these four tools as follows. For each buggy application/kernel in GoBench, its test function serves as the main goroutine. For *goleak*, we insert a call to `defer goleak.VerifyNone(t)` at the beginning of the test function in a buggy application/kernel to detect any deadlock based on its own timeout setting. For *go-deadlock*, we replace `sync.Mutex` and `sync.RWMutex` in a Go program with its own deadlock-aware versions, `deadlock.Mutex` and `deadlock.RWMutex`, respectively. For *Go-rd*, we compile a Go program by turning it on with the "-race" flag. Finally, *dingo-hunter* uses a front-end to translate a Go program into its `MiGo` IR, and then runs a verifier statically on a `.migo` file thus generated. For all the 82 real-world applications in GoReal, its front-end has failed with many exceptions such as "undeclared names". Unfortunately, the front-end does not seem to be capable of searching automatically for the dependencies in large applications (Table III). For the 103 kernels in GoKer, however, *dingo-hunter* has managed to generate the `.migo` files for 45 kernels only.

We have conducted our experiments to measure both the efficiency and effectiveness of these tools for detecting bugs in both GoReal and GoKer. For efficiency, we present our results in Figure 10. For effectiveness, we present our results in Table IV (for detecting blocking bugs) and Table V (for detecting non-blocking bugs, i.e., data races here).

We first present and analyze our results on efficiency and effectiveness and then give a few observations.

A. Efficiency

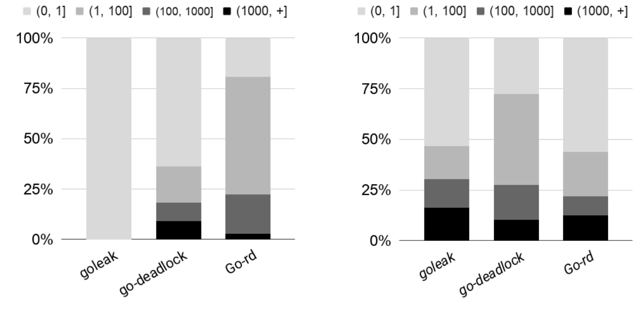
Figure 10 shows the efficiency of the three dynamic tools in finding the bugs in GoReal and GoKer. Let $\mathcal{T}$ be a tool used for finding the bug in a buggy application/kernel $\mathcal{P}$. Given its non-deterministic nature, $\mathcal{T}$ is applied to analyze $\mathcal{P}$ for 10 times. During each analysis, $\mathcal{P}$ is run for up to

TABLE IV
BLOCKING BUGS REPORTED IN GOBENCH ("PRE" STANDS FOR PRECISION, "REC" STANDS FOR RECALL AND "F1" STANDS FOR F1-SCORE).

Suite	Bug Type	*goleak*						*go-deadlock*						*dingo-hunter*					
		#TP	#FN	#FP	Pre (%)	Rec (%)	F1 (%)	#TP	#FN	#FP	Pre (%)	Rec (%)	F1 (%)	#TP	#FN	#FP	Pre (%)	Rec (%)	F1 (%)
GOREAL	Resource Deadlock	1	7	1	50.0	12.5	20.0	7	2	0	100.0	77.8	87.5	-	-	-	-	-	-
	Communication Deadlock	8	13	0	100.0	38.1	55.2	1	16	4	20.0	5.9	9.1	-	-	-	-	-	-
	Mixed Deadlock	3	6	1	75.0	33.3	46.2	4	3	3	57.1	57.1	57.1	-	-	-	-	-	-
	Total	12	26	2	85.7	31.6	46.2	12	21	7	63.2	36.4	46.2	-	-	-	-	-	-
GOKER	Resource Deadlock	14	9	0	100.0	60.9	75.7	23	0	0	100.0	100.0	100.0	0	23	0	-	0.0	-
	Communication Deadlock	20	9	0	100.0	69.0	81.6	0	29	0	-	0.0	-	1	28	0	100.0	3.4	6.7
	Mixed Deadlock	9	7	0	100.0	56.3	72.0	6	10	0	100.0	37.5	54.5	0	16	0	-	0.0	-
	Total	43	25	0	100.0	63.2	77.5	29	39	0	100.0	42.6	59.8	1	67	0	100.0	1.5	2.9

TABLE V
NON-BLOCKING BUGS REPORTED IN GOBENCH.

Suite	Bug Type	*Go-rd*					
		#TP	#FN	#FP	Pre (%)	Rec (%)	F1 (%)
GOREAL	Traditional	23	1	0	100.0	95.8	97.9
	Go-Specific	13	5	0	100.0	72.2	83.9
	Total	36	6	0	100.0	85.7	92.3
GOKER	Traditional	21	0	0	100.0	100.0	100.0
	Go-Specific	11	3	0	100.0	78.6	88.0
	Total	32	3	0	100.0	91.4	95.5



(a) GOREAL (b) GOKER

Fig. 10. Percentage distribution for the (average) number of runs falling into each of the four given intervals that is needed by a tool in finding a bug.

$M = 100,000$ times. Let N_i (where i starts from 1) be the number of runs taken in finding the bug or M otherwise during the i-th analysis. The number of runs needed by $\mathcal{T}$ for finding the bug in $\mathcal{P}$ is $\sum_{i=1}^{10} N_i/10$. Figure 10 gives the percentage distribution for the number of runs falling into each given interval that is required by $\mathcal{T}$ for finding a bug in GOREAL and GOKER.

For `kubernetes#16851` and `docker#27037` in GOREAL, we have used $M = 1000$, since their buggy applications take about 12 seconds and 200 seconds, respectively, in a single run. Every other non-deadlock-related buggy application/kernel takes at most 2 seconds to finish in a single run.

Therefore, dynamic tools can still be rather inefficient, as they must still need to run a Go program multiple times in order to reproduce certain concurrency bugs.

B. Effectiveness

Every buggy application/kernel in GOBENCH has exactly one bug (without any true negative (TN)). Thus, we measure the precision of a tool $\mathcal{T}$ in detecting bugs in GOREAL (GOKER) in terms of the number of true positives (#TP), the number of false positives (#FP), and the number of false negatives (#FN) reported. For a bug in a buggy application/kernel, we have an FN if $\mathcal{T}$ fails to report any bug. If $\mathcal{T}$ does report a bug, we have a TP if the stack trace reported by $\mathcal{T}$ is consistent with the original bug description and an FP otherwise. For *dingo-hunter*, we consider every bug reported as a TP optimistically, since its output is either YES or NO.

1) GOREAL: Let us examine the results in Tables IV and V for GOREAL. According to Table II, GOREAL has 40 blocking bugs and 42 non-blocking bugs. Overall, the four tools are not as effective as expected in finding blocking and non-blocking bugs (measured in terms of precision and recall given).

a) Blocking Bugs: We discuss the results of the three tools in detecting the 40 blocking bugs in GOREAL (Table IV). *goleak* has reported 14 bugs, including 12 TPs and 2 FPs. This tool can find these 12 bugs, since their corresponding test functions have reached developers' exception-handling code due to timeouts (set up by their developers), where the tool has detected some goroutines participating in a deadlock. However, *goleak* has also generated 2 FPs. Finally, *goleak* has missed 26 bugs (i.e., yielding 26 FNs). Among the 26 FNs, 22 of them are because the main goroutine is blocked in a deadlock, thus *goleak* cannot execute any further. The 4 bugs `grpc#1424`, `grpc#2391`, `grpc#1859`, and `kubernetes#70277` are not reported because the developers set timeouts in the buggy function, and the program aborts when timeout exception is detected. *goleak* fails to report those cases since there are no goroutine leaks.

go-deadlock has reported 12 bugs correctly, including 7 resource dseadlocks (5 double locks, 2 AB-BA deadlock), 1 communication deadlock (`cockroach#30452`, where a goroutine is blocked by a buffered channel), and 4 mixed deadlocks (where a lock cannot be acquired in a given period of time, i.e., 30 seconds by default). In addition, *go-deadlock* has also generated 7 FPs: 6 as AB-BA deadlocks incorrectly and 1 due to a lock timeout. Finally, *go-deadlock* has missed 22 bugs, including 2 resource deadlocks (1 due to the timeout of its test function and 1 due to custom locking/unlocking), 16 communication deadlocks (as *go-deadlock* handles neither channels nor the deadlocks related to uninstrumented libraries, such as `context.Context`), and 3 mixed deadlocks (1 due to the timeouts of their test functions and 2 involving `sync.WaitGroup`, which cannot be instrumented by *go-deadlock*).

Finally, *dingo-hunter* fails to find any bug in GOREAL, since it cannot obtain their `.migo` files, as discussed earlier.

b) Non-Blocking Bugs: We discuss the results of *Go-rd* in finding the data races in GoReal (Table V). As shown in Table II, GoReal consists of 42 non-blocking bugs, with 22 data races and 20 other bugs. Note that many non-blocking bugs, such as order violations, also exhibit race-like behaviors, and can thus be detected by a runtime race detector like *Go-rd*. For the results given in Table V, we therefore assume optimistically that *Go-rd* does not produce any false positives.

Of the 42 non-blocking bugs in GoReal, 24 bugs are traditional and 18 bugs are Go-specific (Table II). *Go-rd* has detected all the 24 traditional bugs except a data race bug, `kubernetes#88331` (as the number of goroutines, 8128, has exceeded what can be handled by *Go-rd* [41]). *Go-rd* has failed to detect 5 Go-specific bugs: (1) `serving#4973` and `serving#4908`, due to a panic resulting from misuse of the `testing` library (by calling, e.g. `t.Errorf` to print `testing` logs after the test), (2) `kubernetes#13058`, which is aborted due to misuse of `sync.WaitGroup`, and (3) `grpc#1687` and `grpc#2371`, which are not data-race bugs (by sending a message to a closed channel and a nil channel, respectively).

2) GoKer: Let us examine the results in Tables IV and V for GoKer. According to Table II, GoKer consists of 68 blocking bugs and 35 non-blocking bugs. Again, the four tools are not as effective as expected in finding these bugs (in terms of the overall precision and recall as given).

a) Blocking Bugs: We discuss the results of the three tools in detecting the 68 blocking bugs in GoKer (Table IV). Note that each tool is more effective for GoKer than for GoReal (in terms of precision and recall achieved), as expected.

goleak has found 43 bugs, including 14 resource deadlocks, 20 communication deadlocks and 9 mixed deadlocks, without producing false positives. However, *goleak* has missed 25 bugs, as their test, i.e., main functions are blocked.

go-deadlock has found 29 bugs (without false positives), including 23 resource deadlocks and 6 mixed deadlocks. It has detected these mixed deadlocks due to its timeout mechanism used. For example, `cockroach#1055`, which represents a mixed deadlock involving `WaitGroup`, is found accidentally this way, as is the case of `cockroach#30452` in GoReal. *go-deadlock* has failed to catch 39 bugs, including 29 communication deadlocks and 10 mixed deadlocks (with 2 involving `WaitGroup`).

For the 45 bug kernels that can be compiled by *dingo-hunter*, *dingo-hunter* finds only 1 channel-related communication deadlock, crashes on 29 kernels (due to memory errors and undefined references to the `context` library), and reports no bugs at all in the remaining 15 kernels.

b) Non-Blocking Bugs: We discuss the results of *Go-rd* in finding the data races in GoKer (Table V). As shown in Table II, GoKer consists of 35 non-blocking bugs, including 21 data races and 14 other bugs.

Go-rd has failed in detecting three Go-specific bugs, `kubernetes#13058` , `grpc#1687`, and `grpc#2371`, for the same reasons why it has also failed for the same three bugs included also in GoReal (Section IV-B1b). However, this time, *Go-rd* is successful in finding `serving#4908` in GoKer, since we did not manage to replicate entirely the complex bug-inducing scenario that has caused *Go-rd* to fail in GoReal in the corresponding bug kernel in GoKer.

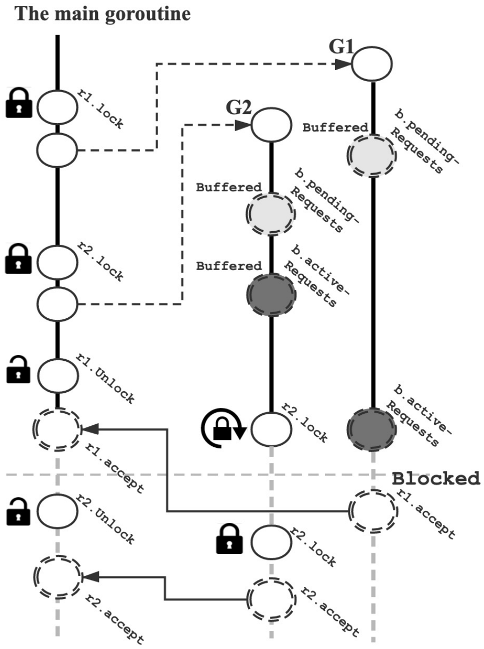


Fig. 11. A blocking bug in `Serving` caused by buffered channels and a mutex (bugID `serving#2137`).

C. Observations

Several observations are in order. First, dynamic tools for Go, which are often based on simple timeout mechanisms, can find the deadlocks and data races in Go programs that are triggered at runtime, making them suitable for continuous integration testing. However, they may find some bugs rather inefficiently (by making excessively many program runs) or never find some bugs at all (by failing to exercise their bug-inducing thread interleavings). Second, static tools are appealing, as they can find potentially the bugs in a Go program without running it. Unfortunately, many language features in Go, such as first-class functions and channels, introduce tremendous challenges to static analysis. In our evaluation, *dingo-hunter*, the only publicly available static tool, performs poorly. Third, model checking techniques [42], [43], which exhaustively exercise all possible message orderings and thread interleavings, are capable of finding more bugs in Go programs. However, for a Go program consisting of often thousands of goroutines, with frequent exchanges of messages, the state-explosion problem faced is daunting. Finally, buffered channels and the multi-choice statement `select` in Go bring non-determinism to a different level. A program may behave very differently when messages are posted to a buffered channel or received by a `select` statement in different orders and timings, making it sometimes very difficult to reproduce a bug. We illustrate this below with a real-world example.

Figure 11 illustrates a bug from the Serving application. The bug involves 3 goroutines (the main goroutine, G_1, and G_2), 2 mutexes (r1.lock and r2.lock), and 4 channels (the 2 buffered channels, b.pendingRequests and b.activeRequests and the 2 unbuffered channels, r1.accept and r2.accept). The main goroutine first acquires r1.lock and forks G_1. Then it acquires r2.lock and forks G_2. Next, it releases r1.lock and waits for the r1.accept message from G_1. G_1 and G_2 have the same entry function and perform the same task. In each goroutine, we first post messages to the two buffered channels via b.pendingRequests and b.activeRequests, and then acquire a lock (r1.lock for G_1 and r2.lock for G_2) to perform a task. After the task is finished, the lock is released and the goroutine posts a message to its assigned channel (r1.accept for G_1 and r2.accept for G_2).

After G_2 has posted to the buffered channel b.activeRequests, if the buffer is full, G_1 is blocked when posting to the same channel. G_2 is also blocked at acquiring r2.lock, which is held by the main goroutine. Since G_1 cannot progress, the main goroutine will wait from channel r1.accept forever, causing a deadlock. In practice, we often need to try tens of thousands of times to trigger the bug. This bug can only be triggered when several events (e.g., 2 locking events and 4 messages here) are processed in a specific order.

To summarize, Go concurrency bugs are challenging to detect. Traditional dynamic detection techniques can help report a bug when the bug can be triggered at runtime. However, there are no good solutions on how to reason about bug-triggering test functions and thread interleavings. We believe GoBench can provide insights on how to tackle this challenging problem.

V. Related work

Studies on the Go language. Tu et al. [9] conduct an extensive empirical study on different types of concurrency bugs in open-source Go applications. Dilley and Lange [8], [9] study how different synchronization primitives are used in Go applications. These studies have motivated our work.

There are several static analysis techniques [44], [45] and verification frameworks [13], [14], [46], [47] for Go applications. Midtgaard et al. [44] introduce a modular approach that can repeatedly analyze one goroutine at a time to reason about communication between goroutines. The developers of *dingo-hunter* [13], [14] model Go programs as communicating finite state machines or a process calculus MiGo to further verify potential communication mismatches. In their later work [46], they have further extended this model to support verification of safety and liveness issues. These approaches are promising but they only provide partial support for the Go language, as demonstrated in our evaluation. GoBench can help researchers evaluate and further improve their approaches.

Benchmarks for concurrency bugs. There are many benchmarks for concurrency bugs in different languages. JBench [48] and DataRaceBench [49] focus on data races in Java. RADBench [50] and JaConTeBe [15] are two benchmark suites of concurrency bugs in real world C/C++ applications and Java applications, respectively. In particular, RADBench provides detailed instructions to reproduce bugs and JaConTeBe provides bug-triggering test cases. TaxDC [51] focuses on concurrency bugs in distributed systems, providing well documented instructions for reproducing each bug. To the best of our knowledge, GoBench is the first benchmark suite of real-world Go concurrency bugs, including real-world buggy applications and bug-triggering test functions in containers to ensure portability. Moreover, we have manually created a large set of small bug kernels, which are representative and straightforward to reproduce and reason about.

Concurrency bug detection. There has been a great deal of research on detection techniques for different concurrency issues. For blocking bugs, resource deadlocks are well studied in the past. Static approaches [19], [21] detect potential deadlocks by checking cycles in lock dependency graphs, which are statically derived from the source code. Dynamic approaches [20], [22] predict potential deadlocks from execution traces. Both static and dynamic approaches suffer from false positives. The work [23] tries to validate reported deadlocks by searching for a bug-triggering interleaving scenario at runtime.

Compared to resource-deadlocks, communication deadlocks are not well-studied. In general, it is undecidable to precisely detect communication deadlocks [52]. Early detection techniques [53], [54] focus on detecting communication deadlocks for MPI-based message passing programs. The researchers in [18] target communication deadlocks caused by condition variables and propose a trace-based model-checking technique to address the problem. Go introduces new features such as buffered channels. Such Go-specific features, compounded further by using channels and locks together, make it challenging to detect communication deadlocks in Go applications.

Traditional race detectors, e.g., ThreadSanitizer [32] and AccuLock [55], detect races at runtime by dynamically computing vector clocks [56] to infer the happened-before relations between instructions. A race condition is raised if no happen-before relation exists between two instructions accessing the same shared memory location (such that one of them is a write). The Go runtime provides a built-in race detector. DataCollider [27] reduces the runtime overhead by sampling, at the cost of potentially missing real bugs. DCatch [57] relies on a set of happens-before rules to model concurrency mechanisms in distributed cloud systems. CloudRaid [58] and CrashTuner [59] target distributed concurrency bugs and crash-recovery bugs for distributed cloud systems, respectively. Despite these extensive studies, concurrency error detection remains open and challenging.

VI. Conclusion and Future work

We have built up GoBench, the first benchmark suite for Go concurrency bugs. Currently, there are 82 real bugs and 103 extracted bug kernels, all based on reported concurrency issues in popular open-source applications. These bugs cover a variety of different types of concurrency issues, especially Go-specific bugs. GoBench is publicly available at *https://github.com/timmyyuan/gobench*. We believe GoBench can help researchers develop practical tools for detecting concurrency bugs in real-world Go applications.

In our future work, we plan to further enrich GoBench by adding more bugs and more bug kernels. We also plan to incorporate some deterministic-replay techniques to make bugs in GoBench easier to reproduce.

Acknowledgment

This work is supported by National Key R&D Program of China (No. 2016YFB1000201), the Foundation for Innovative Research Groups (61521092), the National Natural Science Foundation of China (61802368, 61872043), and Australian Research Council Grants (DP170103956 and DP180104069).

Appendix

A. Abstract

This artifact contains all the material required to reproduce the experimental results reported in our paper titled "GoBench: a Benchmark Suite of Real-World Go Concurrency Bugs". Software dependencies are packaged in Dockerfiles and will be downloaded automatically during the evaluation. You can execute the make command and then validate the experimental results in a PDF file. As our experiments contain many flaky tests, some performance numbers may fluctuate slightly.

B. Artifact Check-List (Meta-Information)

- **Programs:** Go, Python3, all benchmarks programs compiled by us, and all tools evaluated in the paper.
- **Data set:** All benchmarks are included in our sources.
- **Run-time environment:** Docker 19.03.8 or newer.
- **Hardware:** x64 architecture, at least 6 cores and 16G memory are required.
- **Run-time state:** Due to a large number of software dependencies, we recommend experimenting on machines with a network bandwidth of more than 10MB/s.
- **Metrics:** The number of bugs found by each tool, together with some efficiency analysis on the three dynamic tools.
- **Output:** A PDF containing all the experiment results included in our CGO2021 paper.
- **Experiments:** Download the source code and run GNU Make.
- **How much disk space required (approximately)?:** 200 GB.
- **How much time is needed to prepare workflow (approximately)?:** A few minutes.
- **How much time is needed to complete experiments (approximately)?:** 40 hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Data licenses (if publicly available)?:** MIT.
- **Archived (provide DOI)?:** 10.6084/m9.figshare.13257023

C. Description

1) How delivered: The source files can be downloaded at: https://figshare.com/articles/software/gobench-cgo21_tar/13257023

2) Hardware dependencies: Our artifact evaluation requires at least 6 cores and 16GB of memory. In the paper, we have done our experiments on a virtual machine with 1TB SSD, 64GB memory and 16 cores Intel Xeon processors.

3) Software dependencies: The artifact is tested on Ubuntu 20.04, with Docker 19.03.8 and GNU Make 4.2.1.

4) Data sets: We have packaged all the data sets. In the `gobench-cgo21` directory, except for the source code used for running GoBench, we have packaged all the data sets of GoBench in a directory called `gobench`. In more detail, `gobench` is organized as follows:

- The `configures` directory contains the files in json format, recording the type of each bug in GoKer and GoReal and describing how to generate the Dockerfiles corresponding to the bugs in GoReal. These files can be used by researchers and developers to add new bugs.
- The data sets of GoKer and GoReal reside in `goker` and `goreal`, respectively. Under the two directories, each bug is placed in its own directory, which is named like `<project>/<pull id>`. Each bug's own directory contains a `README.md` file to describe the bug.

The experimental startup environments of GoKer and GoReal are packaged in Dockerfiles in the `dockerfiles` directory. All source codes and Dockerfiles changed to experiment with different tools we mentioned in Section IV will dynamically generated during the evaluation and finally output to the `evaluation` directory.

D. Installation

Docker and GNU Make need to be installed in advance. On Ubuntu20.04, run

```
$ apt update && apt install -y build-essential docker.io tar
$ systemctl enable docker
```

For non-root users, you need to make sure that docker can run under non-root privileges [60].

E. Experiment Workflow

To evaluate GoBench, you need to download the archive and unzip it by:

```
$ tar -xvf gobench-cgo21.tar
```

To run all the experiments, you need to enter the `gobench-cgo21` directory and run:

```
$ cd gobench-cgo21/
$ make all
```

These commands finally generate a PDF file named `artifact.pdf` in the same directory.

You can also evaluate GoKer and GoReal separately. To evaluate GoKer, run:

```
$ make goker
```

This command will run *Go-rd*, *goleak*, *go-deadlock* and *dingo-hunter* successively and generate intermediate files ending with `goker.json` in the `result` directory. Their name prefixes should make it clear how these files are related to the results listed in Tables IV and V and Figure 10 in the paper. Similarly, you can use the following command to evaluate GoReal:

```
$ make goreal
```

This command will run *Go-rd*, *goleak* and *go-deadlock* successively. The generated results will be saved in intermediate files ending with `goreal.json`. After having done both, you can run the command below to make a human-readable PDF file (i.e. `artifact.pdf`):

```
$ make pdf
```

`artifact.pdf` will contain Table IV, Table V and Figure 10 in Section IV.

F. Evaluation and Expected Result

All intermediate output log files are stored in the `evaluation` directory. In this directory, each evaluated tool will have its own directory. For example, *Go-rd* will have a directory named `gobench-go-rd` to store log files of experiments for *Go-rd*. The log directories of other tools can be deduced in the same way. Under the directory of each tool, there are two directories `goker` and `goreal`, corresponding to GoKer and GoReal respectively (*dingo-hunter* only has `goker`). In these two directories, each bug still has its corresponding sub-directory.

For each evaluation, the expected results (i.e. Tables IV and V and Figure 10) are stored in the `result` directory, and these data will eventually appear in `artifact.pdf`.

References

[1] Google. (2020) The go language. [Online]. Available: https://golang.org/

[2] Github. (2020) Fastest growing languages. [Online]. Available: https://octoverse.github.com/

[3] Docker. (2020) Docker. [Online]. Available: https://www.docker.com/

[4] CNCF. (2020) Kubernetes. [Online]. Available: https://kubernetes.io/

[5] Google. (2020) The go programming language specification. [Online]. Available: https://golang.org/ref/spec

[6] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[7] Wikipedia contributors, "Communicating sequential processes — Wikipedia, the free encyclopedia," 2019, [Online; accessed 29-February-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=\Communicating_sequential_processes&oldid=929583991

[8] N. Dilley and J. Lange, "An empirical study of messaging passing concurrency in go projects," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 377–387.

[9] T. Tu, X. Liu, L. Song, and Y. 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*, 2019, pp. 865–878.

[10] sasha s. (2016) Online deadlock detection in go (golang). [Online]. Available: https://github.com/sasha-s/go-deadlock

[11] uber go. (2017) Goroutine leak detector. [Online]. Available: https://github.com/uber-go/goleak

[12] fortytw2. (2017) Goroutine leak detector. [Online]. Available: https://github.com/fortytw2/leaktest

[13] N. Ng and N. Yoshida, "Static deadlock detection for concurrent go by global session graph synthesis," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 174–184.

[14] J. Lange, N. Ng, B. Toninho, and N. Yoshida, "Fencing off go: Liveness and safety for channel-based programming," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 748–761, 2017.

[15] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, "Jacontebe: A benchmark suite of real-world java concurrency bugs (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 178–189.

[16] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, 1971.

[17] M. Singhal, "Deadlock detection in distributed systems," *Computer*, vol. 22, no. 11, pp. 37–48, 1989.

[18] P. Joshi, M. Naik, K. Sen, and D. Gay, "An effective dynamic analysis for detecting generalized deadlocks," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 327–336.

[19] K. Sen, D. Gay, M. Naik, and C.-S. Park, "Effective static deadlock detection," in *2009 31st International Conference on Software Engineering (ICSE 2009)*, 2009, pp. 386–396.

[20] P. Joshi, C.-S. Park, K. Sen, and M. 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*, 2009, pp. 110–120.

[21] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter, "Sound static deadlock analysis for c/pthreads," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 379–390.

[22] M. Samak and M. K. Ramanathan, "Trace driven dynamic deadlock detection and reproduction," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 29–42.

[23] Y. Cai and Q. Lu, "Dynamic testing for deadlocks via constraints," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 825–842, 2016.

[24] (2015) Fix kubelet deadlock. [Online]. Available: https://github.com/kubernetes/kubernetes/pull/10182

[25] (2019) sql: fix data race in validatechecks. [Online]. Available: https://github.com/cockroachdb/cockroach/pull/35501

[26] (2018) [galley] fix data race. [Online]. Available: https://github.com/istio/istio/pull/8967

[27] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel." in *OSDI*, vol. 10, no. 10, 2010, pp. 1–16.

[28] L. Chew and D. Lie, "Kivati: fast detection and prevention of atomicity violations," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 307–320.

[29] Q. Zhou, L. Li, L. Wang, J. Xue, and X. Feng, "May-happen-in-parallel analysis with static vector clocks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 228–240.

[30] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," *ACM Sigplan Notices*, vol. 47, no. 1, pp. 387–400, 2012.

[31] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, pp. 37–48, 2006.

[32] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[33] (2020) Data race detector. [Online]. Available: https://golang.org/doc/articles/race_detector.html

[34] C. Labs. (2020) Cockroachdb. [Online]. Available: https://www.cockroachlabs.com/

[35] istio community. (2015) Istio. [Online]. Available: https://istio.io/

[36] (2020) Hugo. [Online]. Available: https://gohugo.io/

[37] (2020) Syncthing. [Online]. Available: https://syncthing.net/

[38] (2020) Knative serving. [Online]. Available: https://knative.dev/docs/serving/

[39] (2020) Etcd. [Online]. Available: https://etcd.io/

[40] (2020) grpc. [Online]. Available: https://github.com/grpc/grpc-go

[41] (2020) failure with limit on … simultaneously alive goroutines is exceeded. [Online]. Available: https://github.com/golang/go/issues/38184

[42] T. Leesatapornwongsa and H. S. Gunawi, "Samc: a fast model checker for finding heisenbugs in distributed systems," in *ISSTA*, 2015, pp. 423–427.

[43] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa *et al.*, "Flymc: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, p. 20.

[44] J. Midtgaard, F. Nielson, and H. R. Nielson, "Process-local static analysis of synchronous processes," in *International Static Analysis Symposium*. Springer, 2018, pp. 284–305.

[45] K. Stadtmüller, M. Sulzmann, and P. Thiemann, "Static trace-based deadlock analysis for synchronous mini-go," in *Asian Symposium on Programming Languages and Systems*. Springer, 2016, pp. 116–136.

[46] J. Lange, N. Ng, B. Toninho, and N. Yoshida, “A static verification framework for message passing in go using behavioural types,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1137–1148.

[47] J. K. Hinrichsen, J. Bengtson, and R. Krebbers, “Actris: Session-type based reasoning in separation logic,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–30, 2019.

[48] J. Gao, X. Yang, Y. Jiang, H. Liu, W. Ying, and X. Zhang, “Jbench: A dataset of data races for concurrency testing,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 6–9.

[49] P.-H. Lin, C. Liao, M. Schordan, and I. Karlin, “Exploring regression of data race detection tools using dataracebench,” in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2019, pp. 11–18.

[50] N. Jalbert, C. Pereira, G. Pokam, and K. Sen, “Radbench: A concurrency bug benchmark suite.” *HotPar*, vol. 11, pp. 2–2, 2011.

[51] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, “Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 517–530.

[52] D. Brand and P. Zafiropulo, “On communicating finite-state machines,” *Journal of the ACM (JACM)*, vol. 30, no. 2, pp. 323–342, 1983.

[53] S.-T. Huang, “A distributed deadlock detection algorithm for csp-like communication,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 102–122, 1990.

[54] S. Chen, Y. Deng, P. Attie, and W. Sun, “Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs,” in *Proceedings of 16th International Conference on Distributed Computing Systems*. IEEE, 1996, pp. 613–619.

[55] X. Xie and J. Xue, “Acculock: Accurate and efficient detection of data races,” in *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 2011, pp. 201–212. [Online]. Available: https://doi.org/10.1109/CGO.2011.5764688

[56] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications*, 1978.

[57] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, “Dcatch: Automatically detecting distributed concurrency bugs in cloud systems,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 677–691, 2017.

[58] J. Lu, F. Li, L. Li, and X. Feng, “Cloudraid: Hunting concurrency bugs in the cloud via log-mining,” ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–14. [Online]. Available: https://doi.org/10.1145/3236024.3236071

[59] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You, “Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 114–130. [Online]. Available: https://doi.org/10.1145/3341301.3359645

[60] D. Inc. (2020). [Online]. Available: https://docs.docker.com/engine/install/linux-postinstall/