2022

How Many Mutex Bugs Can a Simple Analysis Find in Go Programs?

Fumi Takeuchi
Tokyo Institute of Technology

Hidehiko Masuhara
Tokyo Institute of Technology

Raffi T. Khatchadourian
CUNY Hunter College

Youyou Cong
Tokyo Institute of Technology

Keisuke Ishibashi
International Christian University

How many mutex bugs can a simple analysis find in Go programs?

Fumi Takeuchi   Hidehiko Masuhara   Raffi Khatchadourian

Youyou Cong   Keisuke Ishibashi

In open source software, it is known that there are many concurrency bugs. A previous study in Go revealed that a considerable number of such bugs are simple (for example, 9% of the bugs are the ones that forget to unlock a mutex,) through a manual program investigation. This paper tries to detect such bugs by applying a simple analysis in order to see how far such a tool can match the manual analysis. We built a simple intraprocedural control flow analysis in Go, and evaluated its performance with respect to the open source programs with concurrency bugs reported in the previous study. Consequently, as for quality, the recall is good at 88% and the precision is poor at 60%, and as for analysis time, it can be finished within practical amount of time (for example, 1 second per 5000 LoC).

1 Introduction

Forgetting to unlock mutual exclusion (hereinafter referred to as mutex) is one of the major root cause of bugs dealing with mutex. This bugs will lead to deadlock errors by waiting for resources forgotten to be released which will never be available. We call this bugs as mutex bugs, and forgotten unlock as missing unlock.

Mutex bugs sometimes lead to serious and fatal errors in software. For example, [27] is a mutex bug found in MariaDB recently which will cause denial of service fatal error. The cause of [27] was that acquired mutex was not released correctly in the error handling case.

Among many languages, the Go programming language (hereinafter referred to as Go) programs seems to have many mutex bugs. Like the case of [27], mutex bugs can be found around error handling, since programmers need to unlock mutex at both of the main functionality and error handling. If a language has the feature of `try-catch-finally` exception handling, programmers can put unlocks together in `finally` clause. Nevertheless, Go intentionally omit this style exception handling system [9].

Previous study [48] shows that considerable amout (about 9%) of bugs which will lead to deadlock error are mutex bugs. This study was conducted by the manual investigation on real-world open source software (hereinafter referred to as OSS) written in Go.

We aim to propose a mutex bugs checker to see how frequent those bugs appear in real world Go programs. In this paper, mutex bugs checker made with control flow analysis is shown in Section 3

and its evaluation regarding the quality and the performance is illustrated in Section 5.

Previous studies enable the detection and automated fix, by using session graphs [43], behavioral types [38][39], novel constraints solver [41], modeling Go's communication between processes [45], none of the above studies did investigate the possibility of a simple analysis, whose analysis could be finished faster.

2 Background

2.1 Go and Concurrency

To deal with concurrency, it is essential to consider how to synchronize each process at some point. Concurrent programs consist of two parts, one is the point that is needed to run synchronously, and the other is the point that can be run simultaneously since each process does not depend on the other [42]. The latter one is called embarrassingly parallel, and for concurrent programs, the method to deal with the former is significant. In fact, many techniques are proposed to turn the former part into the embarrassingly parallel, and it is better to do so if it can be, but in Go, the former case can be seen a lot, so this paper focuses on that.

In Go, there are two commonly used methods to deal with concurrency. One is mutual exclusion (hereinafter referred to as mutex), and the other is channel. Mutex is a well-known method often used in other languages while channel is not.

2.2 Mutex

Mutex is a well-known and low-level mechanism for concurrency to exclude race conditions. Race condition happens when concurrent processes operate the same memory without synchronization and if one of them is the write operation [42]. For example, suppose there are one global variable V and two processes executed concurrently A and B

which increment V. Here, increment operation is done by loading from the memory, adding 1 to the value, and then writing the result to the memory. The expected result value of V is 2 since it is incremented two times. However, A and B are done concurrently, it is possible that loading in A is done between loading in B and writing in B. In this case, both A and B write 1 to the V, and this is called a race condition. Troublesomely, race condition does not happen every time. In the previous example, it is also the case that A is done after the writing operation of B, which results in no race conditions. Thereby, to eliminate the possibility of race conditions, mutex is used. According to [42], mutex has two states, locked and unlocked, and two operations, lock and unlock. Each operation is done atomically, thus a process has to wait to possess the lock of mutex from locked mutex by the other process to be unlocked. Therefore, for the above example, by wrapping lock and unlock to the same mutex object each increment in A an B, the race condition is resolved and always the result V becomes 2. Besides, the part which must be protected due to race condition is called a critical section.

However, since processes cannot go through the critical section until the lock is released, if one process forgets to release the lock, every process cannot proceed. This is called a deadlock error. Actually Go has two types of deadlocks: one is a global deadlock, and the other is a partial deadlock. Global deadlock means all the processes are halted, on the other hand, partial deadlock indicates some of the processes are halted.

2.3 Channel

Channel is like a stream with buffers to communicate between processes. Based on Hoare's Communication Sequential Processes [34] (here-

inafter called CSP), channel is brought to Go. As Hoare described in [34], input and output between processes are the primitives of communications, and he illustrates communication as send and receive values between processes. Originally in [34], transceiving is done directly from and to processes, however, he also considered communicating through a port, a sort of variable for input and output like a stream.

2.4 Go and Static Analysis

In Go, static code analysis tools are often used. One example is a lint tool called `go fmt` (or `gofmt`) [8]. Go officially provides `go fmt` command for the sake of formatting codes. Thanks to this static analysis tool, the number of conflicts over coding styles might be fewer than other languages since Go programmers can delegate it to the standard [46]. The other example is `go vet` [18]. As the description in [18] says, `go vet` command reports common mistakes in a heuristical way by analyzing code statically.

2.5 Control Flow Analysis

To analyze programs statically, it is needed to know in what order the program will be executed. To have this information, Control Flow Graph (hereinafter called CFG) represents the order of the program execution.

CFG is a directed graph that consists of basic blocks which are composed of a linear sequence of a fragment of a program [28]. In short, a part of a program whose order of execution depends on condition is expressed as a path, while a part of a program that has immutable order is expressed as a vertex. The example code in Go and its CFG are shown in Listing 1 and Figure 1.

Listing 1 Example Code for CFG

```
1  // F returns the position of 0
2  // in the given lisrt.
3  // If no 0 in list, then return
4  // the length of the list.
5  func F(list []int) int {
6    fmt.Println("start")
7    counter := 0
8    for _, l := range list {
9      fmt.Println("check", l)
10     if l == 0 {
11       fmt.Println("0 found", counter)
12       break
13     }
14     counter++
15   }
16   fmt.Println("end")
17   return counter
18 }
```

The code in Listing 1 contains some conditional branches. On line 6, if iteration over `list` has not done, the body of `for` loop (hereinafter called *range.body*) will be executed. Otherwise after line 14 (hereinafter called *range.done*) will be executed. And on line 8, there is a `if` statement dependent on the condition of the current `list` item `l` in the iteration. If the condition holds, `break` will be executed subsequently, so the successor of this `if` body (hereinafter called *if.then* is *range.done*.

And Figure 1 represents the CFG of Listing 1. Each node consists of the fragment of linearly sequential processes [†1].

2.6 Strongly Connected Components

In a directed graph, if the subgraph of it consists of vertices that are reachable from any other vertices, the region is estimated as strongly con-

---

†1 Depending on the implementation of CFG builder, the computation for conditions (such as `l == 0` in the Listing 1) are sometimes included in successive nodes (here means 4 *if.then* instead of 2 *range.body*), but in this paper, regarding it as included in its predecessor.
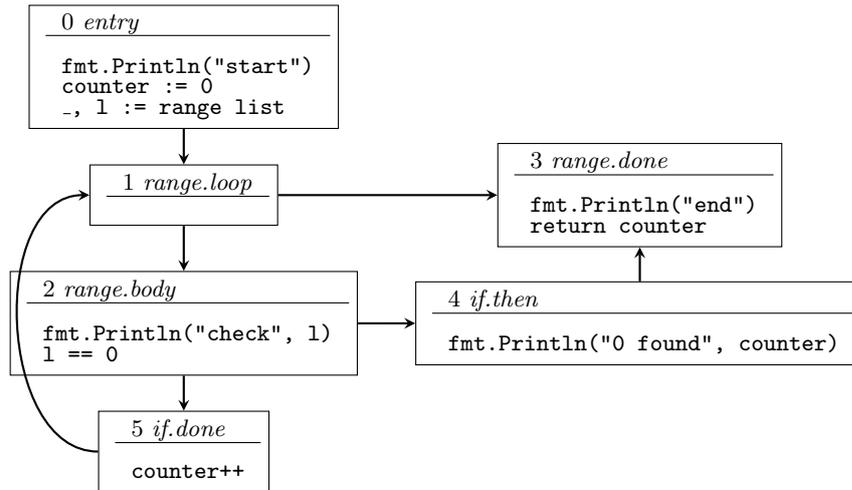
図 1 CFG of Listing 1

nected [28]. This is important for analysis since this represents a loop in CFG. In the above example Figure 1, the components which can be estimated as strongly connected are 1, 2, and 5, as shown in Figure 2 with enclosed by the red dashed frame.

By seeing strongly connected components (hereinafter referred to as SCC) as a block, this graph can be assessed as a directed acyclic graph (hereinafter called DAG).

3 Methodology

To check mutex bugs, our checker attempts to find missing unlocks, in other words, the position that locks may be called without unlocked. Listing 2 shows the Go code contaning mutex bugs, while Listing 3 is the fixed version of Listing 2. The Listing 3 illustrates three patterns of the missing unlock positions.

Listing 2 Go Code with Mutex Bugs

```
1  func _(mu *sync.Mutex, cond bool, list []
       int) {
2    mu.Lock()
3    if cond {
4      return
5    }
6    mu.Unlock()
7
8    for _, l := range list {
9      mu.Lock()
10     if l == 0 {
11       continue
12     }
13     if l == 1 {
14       break
15     }
16     mu.Unlock()
17   }
18 }
```

Listing 3 Fixed code of Listing 2

```
1  func _(mu *sync.Mutex, cond bool, list []
       int) {
2    mu.Lock()
3    if cond {
4  +    mu.Unlock()
5      return
6    }
7    mu.Unlock()
8
9    for _, l := range list {
10     mu.Lock()
11     if l == 0 {
```
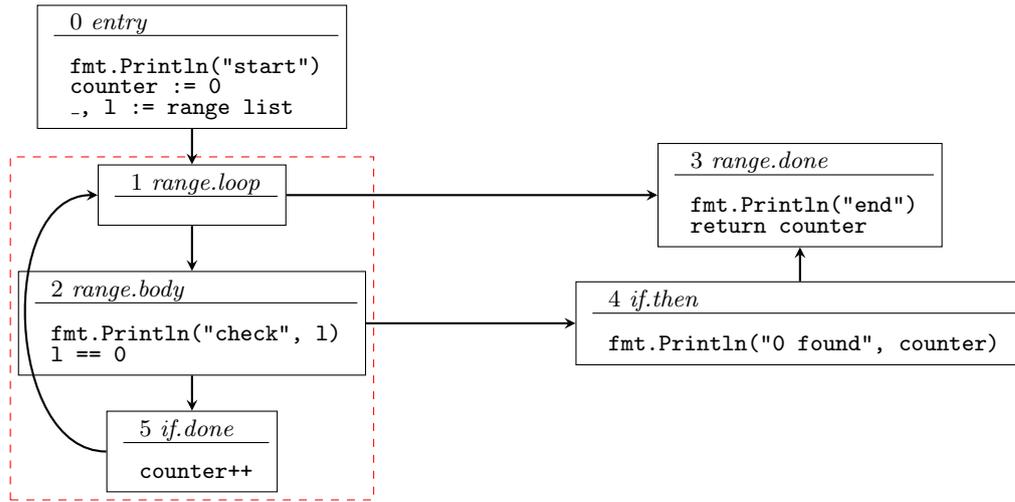
図 2 CFG of Listing 1 with a Red Dashed Frame of Strongly Connected Components

```
12 +      mu.Unlock()
13        continue
14      }
15      if l == 1 {
16 +      mu.Unlock()
17        break
18      }
19      mu.Unlock()
20   }
21 }
```

To check mutex bugs, firstly CFG built on each function declaration including methods and anonymous functions. Since the analysis strategy is intraprocedural, CFG builder does not trace function calls inside function declaration. In brief, the checker interprets a project as a cluster of many CFGs. After CFGs are constructed, the checker traverses each CFG by DFS. At each DFS step, lock and unlock operations for each mutex object are recorded (corresponding to `lockstate` in Listing 4). Then current CFG block is determined as containing mutex bugs or not (corresponding to `report` procedure call in Listing 4). The whole specification of the procedure is shown in Listing 4.

Listing 4   Pseudo code of checker's common algorithm by DFS

```
1 Require:
2   block: CFG's block,
3   lockstate: hashmap of boolean,
4   report: procedure which require block and
            lockstate,
5
6 procedure DFS(block, lockstate, report)
7   for each node in block's AST nodes
8     if node is an operation to a mutex
          object
9       mu = the operated mutex object
10      if operation is lock
11        lockstate[mu] = true
12      else if operation is unlock
13        lockstate[mu] = false
14      end if
15    end if
16  end for
17
18  report(block, lockstate)
19
20  for each succ of block
21    DFS(succ, copy of lockstate, report)
22  end for
23 end procedure
```

The report function tries to find three types of

patterns, one is missing unlock reporter before return, another is the checker for lock-unlock consistency in a loop, and the other is the missing unlock reporter before exiting from a loop by `break`. These reporters are run sequentially and sometimes report the same missing unlock. If the reports are the same suggestion, it must be reported only one of them.

If any mutex objects' lockstate are true and the block does not have any successors, it should be reported at the block since it means releasing of the mutex object's lock is not done before exiting from the function. For example, in Listing 5, on line 4, the mutex object `mu` has been locked but not released before exiting from the function.

Listing 5   Missing Unlock Before Return

```
1 func _(mu *sync.Mutex, cond bool) {
2   mu.Lock()
3   if cond {
4     return
5   }
6   mu.Unlock()
7 }
```

And in Figure 3, the CFG of Listing 5 is shown. As it indicates exiting block (block 1 and 2) from the function does not have any successors, so it can be reported. In those two, block 2 does not have unlock call for `mu`, therefore it must be reported. Additionally for the automated fix, unlock call for `mu` must be inserted at the last position before `return` of AST nodes in block 2.

If any mutex objects' lockstate is true, the current block's successor is a loop entrance (like `range.loop` or `for.loop`), and the lock is acquired in the SCC which includes the loop entrance, it must be reported. For instance, Listing 6 and corresponding CFG given in Figure 4 contains a missing unlock in `if.then` block, since on line 5 `continue` is executed without unlock for `mu`. If
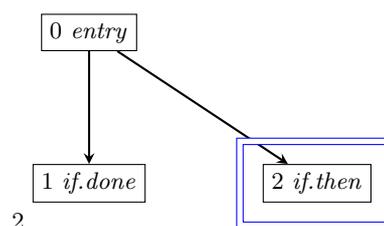

図 3   CFG of Listing 5 with a Blue Doubled Frame For the Block to be Reported (corresponding AST Nodes are abbreviated)

`continue` is executed, acquirement for the lock of mu on line 3 will be executed, although it causes deadlock since it is not unlocked.

Listing 6   Missing Unlock in the Last of Loop

```
1 func _(mu *sync.Mutex, list []int) {
2   for _, l := range list {
3     mu.Lock()
4     if l == 0 {
5       continue
6     }
7     mu.Unlock()
8   }
9 }
```

If any mutex objects' lockstate is true, the current block is the bridge from an SCC, and all the predecessors of the successors of the current block except the current block are in the same SCC, it must be a break from a loop without unlocking. As an example, Listing 7 and its CFG in Figure 5 represents the error captured by this algorithm. On line 5 `break` from a loop is executed without unlocking the `mu` mutex, therefore if `mu` lock is tried to be acquired in other functions or somewhere in the same function, it will be a deadlock error.

Listing 7   Missing Unlock in the Last of Loop

```
1 func _(mu *sync.Mutex, list []int) {
2   for _, l := range list {
3     mu.Lock()
4     if {
```
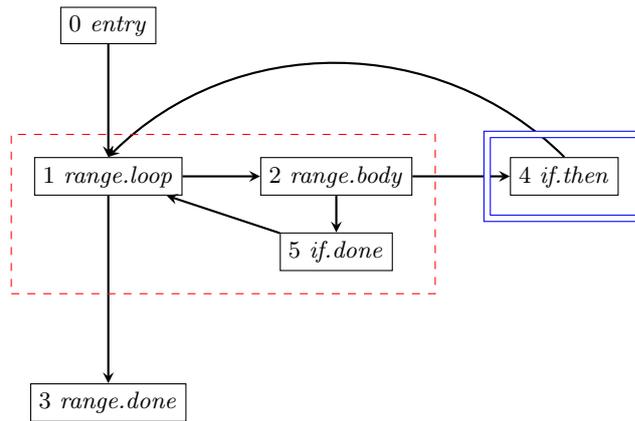
図 4 CFG of Listing 6 with a Red Dashed Frame of Strongly Connected Components and a Blue Doubled Frame For the Block to be Reported (corresponding AST Nodes are abbreviated)

```
5      break
6    }
7    mu.Unlock()
8  }
9 }
```

4 Implementation

To implement simple analysis checker and automated fix tool in Go, commonly `analysis` package [1] is used as mentioned in Section 2. Additionally, previous studies use external languages or tools like C++ [41] or Haskella and mCRL2 [39][44], but this paper only uses Go and Go libraries to make the installation process as easy as possible.

The implementation will be available at `https://github.com/prg-titech/mutexunlock`.

4.1 Control Flow Graph Builder

Under [1], some helpful checkers are provided by the Go team semi-officially. For the CFG Builder. Among those, `ctrlflow` package [5] helps to build CFG from the source code. This package constructs CFG based on intraproceduraal analysis, which means panic outside of a function will be omitted from CFG.

However, [5] does not have the position information corresponding to blocks. AST nodes in blocks know their position in source code, but if a block does not have any AST nodes, there is no way to identify the block's position, though the position data are required to report or insert release calls to developers.

For this reason, we decided to fork the `ctrlflow` package [5] to supplement position information to blocks. By virtue of this fork, for example, report of break like in Listing 7 was realized, since without position data the `if` then block have no AST nodes which are required to report.

In terms of the limitation of using this package, all the code must be buildable before static analysis. This is required to build the type and value information for variables and functions.

4.2 Fix Suggestion

[1] also has fix suggestion system. By putting the start position, the end position, and the alternative text to `TextEdit` [2] struct, fix suggestion could be implemented. To use, the only requirement is to add `-fix` option when executing the analyzer binary.
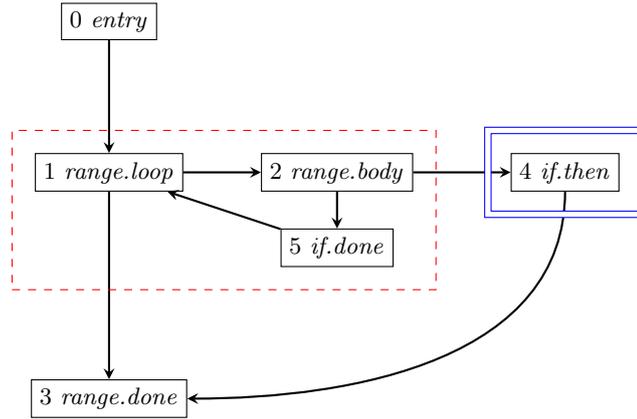
図 5   **CFG of Listing 7 with a Red Dashed Frame of Strongly Connected Components and a Blue Doubled Frame For the Block to be Reported (corresponding AST Nodes are abbreviated)**

4.3   Strongly Connected Components

To know whether the nodes are strongly connected or not, this implementation used Tarjan's algorithm [47] to construct SCC data because of its efficiency.

5   Results

We evaluate our checker's feasibility with respect to real-world code by applying the checker to selected open source software projects that are already reported to have mutex bugs in the previous study. The selected projects are namely BoltDB [3], CockroachDB [4], Docker [16][†2], etcd [7], gRPC-Go [12], and Kubernetes [15] on GitHub.

First, from the bugs reported in the previous study [48], we selected the ones that are related to mutex bugs. The bug-fixing commits are retrieved from the previous study [48]. They collected many fix commits related to concurrency bugs from popular OSS. The number of such bug-containing commits is 8.

Next, for each bug, we apply the buggy commit to the detector, where a buggy commit is one version prior to the fix commit of the bug reported in

the paper [48].

Finally, by running the detector on those codes, the results are obtained. The quality is calculated based on recall and precision of the results, and the performance is obtained by calculating mean response time and its SD of 10 times running by `time` command in GNU Bash [10] for each commit.

All the experiments are done on an arm64 Ubuntu 20.04 docker (runsc [11]) instance running on a VM.Standard.A1.Flex Oracle Cloud instance, whose shared CPU is 4 core Ampere(R) Altra (Neoverse-N1) and memory is 24GiB.

As a result, Table 5 shows the overall consequence for the quality and the performance of the checker with the all or target module source code statistics. While each row represents the bug-containing commit and the applied results with the lines of code (hereinafter called LoC), the number of functions, the number of blocks, and the number of edges included in the module, each column shows the following statistics:

- "Runnable" represents whether the checker can be run.
- "Detected" shows how many missing unlock was reported.

---

†2   Docker repository is renamed to Moby

- "False Positives" illustrates the number of false-positive reports
- "True Positives" indicates the number of reasonable reports.
- "Manually Fixed in [48]" column implies how many missing unlock was fixed manually by the programmer originally, in other words, the number of bugs expected to be reported.

If the checker is run on a module, the lines of code (hereinafter called LoC), the number of functions, the number of blocks, and the number of edges are the code included in the module, not a whole project. Due to the limitation of the implementation, the number of functions, the number of blocks, and the number of edges cannot be obtained if the target source code is not buildable so that those fields of the commit 8eba018 in Docker [20] are empty.

表1 Results of the Evaluation with Characteristics of Each Project, Including the Number of Detected Missing Unlock, the Number of False Positives and True Positives, and the Number of Manually Fixed Bugs in [48] for Each Commit.

Project	Commit	Module*	LoC	Number of Functions	Number of Blocks	Number of Edges	Runnable	Detected	False Positives	True Positives	Manually Fixed in [48]	Mean Response Time (s)	SD (s)
CockroachDB	a889f82 [21]	gossip	513	89	552	489	△¹	5	2	2(3)**	2**	2.14	0.15
CockroachDB	1f70b73 [23]	(Project)	118,783	4,527	49,482	47,174	✓	5	4	1	1	20.06	0.27
CockroachDB	d7bb465 [25]	util	24,415	799	5,779	5,048	△²	1	1	0	0	2.57	0.25
Docker	6457f5 [19]	proxy	512	20	106	85	△²	1	0	1	1	1.13	0.18
Docker	8eba018 [20]	(Failed)	303,110	-	-	-	✗	-	-	-	(1)***	-	-
gRPC-Go	7c6103d [24]	(Project)	24,200	1,121	5,888	4,734	✓	4	3	1	1	2.78	0.33
Kubernetes	2e7993e [22]	storage	5,279	134	1,037	956	△³	2	1	1	1	4.35	0.22
Kubernetes	07424af [26]	cloudprovider	25,307	817	7,437	6,673	△³	2	1	1	1	6.95	0.26
Total								20	12	7(8)**	7(8)***		

¹ Runnable only on subdirectory due to memory usage
² Runnable only on subdirectory due to build error
³ Runnable only on sub-sub directory due to build error and memory usage on subdirectory
* (Project) means the analyzer could be run on the whole project (the root path), (Failed) means the analyzer could not be run, and the others are the sub or sub-sub directory (module) names which the analyzer could be run.
** The detector found a unknown missing unlock not fixed in [48]
*** Since the detector cannot be run on Docker 8eba018, it is excluded from the sum.

According to Table 5, for the quality, the recall is 88% (7 out of 8) while the precision is 60% (12 out of 20). For the performance, it took 1 second to analyze 5000 LoC in this experiment with around 1-second start-up time. The plot is shown in Figure 6.

As Figure 6 indicates, the relationship between LoC and the time taken by applying to the target source code indicates that the time is also linearly increased as the LoC rises.

6 Discussion and Future Work

6.1 Discussion

First of all, the rate of missing unlock is 8 out of 83 blocking bugs, which denotes around 9%. This rate accords with the previous study results, as [41] shows that the missing unlock bug is approximately 13% of all the bugs they used in their experiment. Although it is not such a high rate, considering that these are caused by careless mistakes that language design could eliminate, the fact that missing unlock is fixable using the detector cannot be ignored. Moreover, blocking bugs halt the whole of the program suspiciously despite the difficulty of reducing the state complexity, so if 9% of the blocking bugs can be reduced, it is meaningful.

Among them, Section 5 shows the simple analysis algorithm can detect and fix automatically such missing unlocks at a high rate, 100% for the detection and the automated fix with the exception of the unbuildable case (if not included, 88% for the detection and the automated fix). Although the number of bugs is not so much compared to previous work [41], the fact that such a simple detector demonstrates detection and fix at a high rate is significant.

However, the number of false positives is considered to be a problem. Although according to Section 5, false positives are 12 out of 20 which mean 60% of the whole reports, obviously it will depend on the code size and the domain of modules, as the results of CockroachDB's 1f70b73 [23], and gRPC-Go's 7c6103d [24] in Table 5 imply that reports on the whole project might have high rate false positives at 75% to 80%. the report by "Missing Unlock Before Exiting from a Function" for Intentionally-separated lock and unlock functions, and regarding a block as break block mistakenly by "Consistency After Breaking from Loop" commit the high rate of false positives.

The rate of false positives has a tendency to be high on the whole project running. The results of CockroachDB's 1f70b73 [23], and gRPC-Go's 7c6103d [24] show 4 out of 5 and 3 out of 4 respectively, which means 75% to 80% of the reports. Mainly, those false positives reports consist of separate function design for lock and unlock like transaction's begin and commit function, in other words intentionally not to unlock at the exit point of a function. The other reports are caused by the analysis algorithm. Listing 8 shows the false-positive report due to the detection algorithm.

Listing 8   A False Positive Fix Suggestion in gRPC-Go

```
1  diff --git a/balancer.go b/balancer.go
2  index 419e2146..36bdd220 100644
3  --- a/balancer.go
4  +++ b/balancer.go
5  @@ -357,6 +357,7 @@ func (rr *roundRobin)
       Get(ctx context.Context,
       opts BalancerGetOptions) (addr Ad
6  ...
7  for {
8  ...
9    if rr.waitCh == nil {
10     ch = make(chan struct{})
11     rr.waitCh = ch
12   } else {
13     ch = rr.waitCh
14 +   rr.mu.Unlock()
15   }
16   rr.mu.Unlock()
```
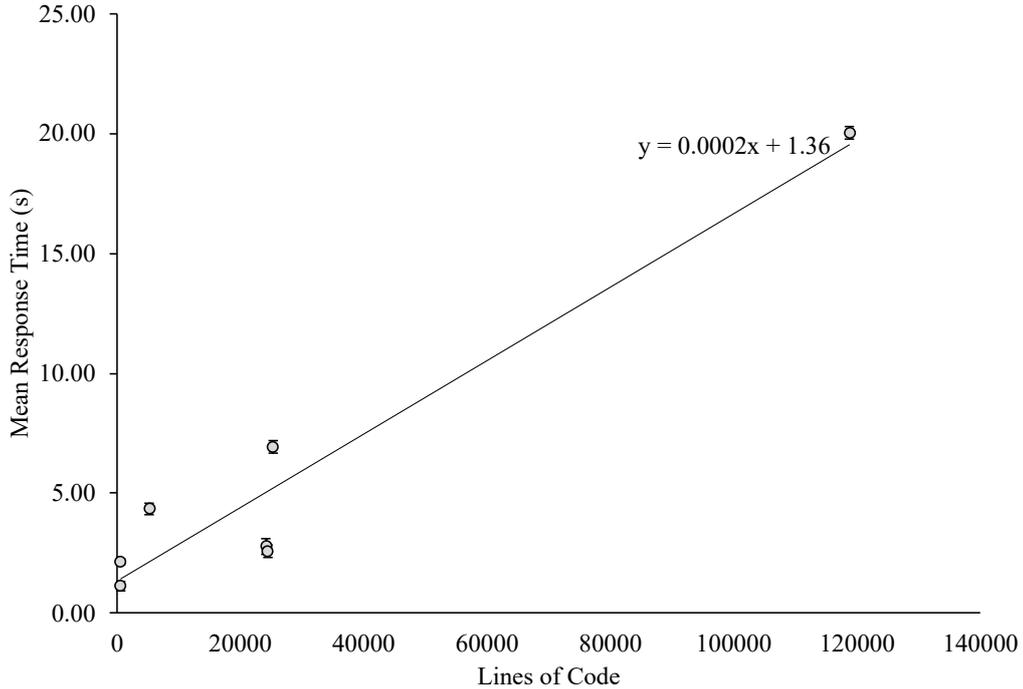
図 6 **The Relationship between Response Time of the Analyzer by Applying to Each OSS and Lines of Code with *SD* Error Bar**

In Listing 8, unlock is done after the else clause therefore it is not needed to insert `rr.mu.Unlock()` in the else clause. This fix is suggested by the "Consistency After Breaking from Loop" reporter since the algorithm misunderstands the else clause as the block which contains a break because of the CFG shape. False-positive reports can be divided into these two types.

For benchmark, Figure **??** indicates that the detector running time is linearly increased along with the amount of the sum of the number of blocks and number of edges. This result meets the original estimation described in Section 1 and Section 3. Interestingly, Figure 6 indicates that the detector benchmark shows the linear increase with LoC, which is a more suitable indicator for the scale of a project because of its ease of measurement.

6.2 Future Work

6.2.1 Algorithm

The current algorithm used in "Consistency After Breaking from Loop" has a problem to determine whether a block represents a break from a loop or not. The cause of this problem is using SCC to know whether it is a loop or not. Newer methods to identify the loop are proposed like [50], thus using these instead of SCC will solve this problem.

Additionally, "Missing Unlock Before Exiting from a Function" is the one that commits to false positives. One simple solution to the issue is that firstly the just lock functions are marked as candidates to be reported, and then if those are not con-

sistent in the callee function, it can be reported.

6.2.2 Channel Support

Currently, the tool does not support channels, so it must be supported in the future. Actually, previous work [41] can detect and automatically fix it. Details will be described in Section 7.

6.2.3 Requirement for Full Builds and Memory Usage

`ctrlflow` package [5] requires the target source code to be buildable, which is not needed for the construction of CFG. However, type information for variables is necessary for detecting mutex object. Currently, we could not find a great way to deal with this requirement. On the side of memory usage, 3 cases were failed due to the high memory usage during the analysis. This is caused by `ctrlflow` package [5], not our implementation part. In [35] it looks runnable on relatively large projects which is failed in this study due to memory usage error, therefore maybe using other package to build CFG would solve this problem.

7 Related Work

Bugs related to concurrency were examined a lot. They can be roughly divided into three types: one is the study about bugs itself, another is the bug detection, and the last one is the automated bug fix.

7.1 Study on Concurrency-Related Bugs

[48] is the study which collects bug fixes related to concurrency bugs from popular OSS in Go. As Go introduces the different approach from other languages to communicate between processes, which is channel, they studied how channel impacted the number of concurrency bugs. The result is that more than half of 171 bugs in total are caused due to Go's characteristics. They also mentioned that by using channel, the number of shared memory synchronization bugs, which is treated in this paper, is decreased. In contrast, the number of bugs related to channel is increased.

Likewise, in other languages, concurrency-related bugs in OSS are also researched as the study on real-world bugs investigation. [49] is one of the examples in Node.js. They collected and analyzed 57 bugs among popular OSS in Node.js, and as a result, they found two-thirds of them are atomicity violations due to the lack of language and runtime support of locks and transaction mechanisms.

7.2 Concurrency-Related Bugs Detection

For bug detection, there are dynamic and statical ways. Go provides a dynamic race condition bug detection tool officially [6]. Additionally, Go can detect global deadlock errors at runtime since it can be investigated by observing processes and timers. If none of them has progress, it means the program faces a deadlock error. However, dynamic detection cannot know the bug until they meet, therefore sometimes edge case bugs remain in a released software emerged as investigated bugs in [48].

Therefore statical analyses are essential, though it is not provided officially. [31] illustrates the current overview of statical concurrency bug detection studies in Go. The first one which proposed the statical concurrency bug detector in Go [43] used session graph. From the same research group, two research based on their single static assignment intermediate representation and its behavioural types checking [38] [39] are published. Moreover, they used bounded model checking in [32] to aim for supporting more large scale source code. The verification of communicating session automata for concurrency is discussed in [40]. One

more study with Go and behavioral types is [33]. This paper extends the usage to not only deadlocks but also race condition detections. Another approach is proposing Mini-Go, a subset language of Go only for concurrent features with its formalization [45]. They used it to detect global deadlock errors. In addition to those studies, [30] made a transpiler from Go to Coq and applied the code to model checker Iris [14] which provides concurrency proofs. Likewise, concurrency proofs are encouraged in other languages than Go. For example, [38] uses Haskell implementation termination checker and mCRL2 [13].

7.3 Automated Fix for Concurrency-Related Bugs

Concurrency-related bugs detection and fixes are on dependence relation; it must be detected at least to fix bugs.

[41] is a direct previous work published in 2021, which uses a similar implementation to this paper. Firstly they modeled channel and its communication between processes. Mutex can be expressed as channel with a size 1 buffer they said, therefore modeling is enough to support the fix for full Go's concurrency. Secondly, all the possible path combinations are calculated. Then, they used a novel constraint solving method (using Z3 [29]) to examine buffer size overflow for each possible path. Contrary to this paper, the path combination can beyond the unit of function.

In other languages, for example in C, while [35] provides automated fix using XXX, [37] fixes automatically by a constraint solving especially for mutex lock and unlock in C.

As the other example in Java, [36] uses typestate analysis to examine Stream API which was introduces in Java 8 [17]. This study is different from others a little, in terms of its responsibility is not fix but refactoring, because refactoring implies the aspect of alleviation.

8 Conclusion

Bugs in software lead to critical problems occasionally. Especially, concurrency-related issues are relatively more complicated to fix among those bugs due to the complex states rather than those sequential programs'. Notably, the combination of mutex for data race avoidance and error handling without exception handling makes it harsh to manage mutual exclusion unnecessarily; nevertheless, the worst-case scenario halts the whole program.

Besides, Go is a language advertised for concurrency support without exception handling. In fact, such mistakes can be found in the list of real-world bugs [48] in Go. Seemingly, these bugs in Go can be detected by using the analysis on the CFG, therefore this paper discussed such a detector really works by applying to the real-world fixes collected in [48].

For the detector, we focus on three missing mutex releases, one is before exiting a function, another is before going back to a loop's head, and the other is breaking from a loop. At each step of DFS to traverse CFG, the lock state for each mutex object is stacked, and if the latest state holds one of the three conditions at least, it will be reported at the block in the CFG. At the same time, fix candidates are also suggested by inserting unlock call for the mutex object at the position reports emerged.

Meanwhile, the detector is evaluated by applying those fixes in OSS in [48]. Firstly, those fixes which are suspected to be missing mutex unlock picked up manually from blocking bugs list in [48]. Secondly, the detector is applied to one before commit of the target fix. Finally, the evaluation is done by comparing the detection and fix reports

with the original programmers' fixes.

As a result, there are 8 out of 83 (approximately 9%) satisfying bugs. This result is reasonable considering other previous work like [41]. The detector can appropriately report and fix all of them except for 1 build error project. On the other hand, the rate of false positives is 75% to 80%, which is relatively high especially for large LoC targets.

In conclusion, although the number of missing unlock fixes are not so large, it can be stated that if 9% of bugs ascribed to trivial mistakes can be eliminated, it is worth applying. However, the rate of false positives is rather problematic. To decrease the rate, the analysis algorithm must be extended, like marking at call and checking at callee. Moreover, the determination algorithm for the loop must be replaced by other techniques like [50]. Overall, through this study at least it can be shown that the simple detector made with CFG analysis can detect simple missing unlock.

9 Acknowledgment

参 考 文 献

[1] : analysis package, `https://golang.org/x/tools/go/analysis`. Accessed: 2021-12-21.

[2] : analysis package - golang.org/x/tools/go/analysis - pkg.go.dev, `https://pkg.go.dev/golang.org/x/tools/go/analysis#SuggestedFix`. Accessed: 2022-01-31.

[3] : boltdb/bolt: An embedded key/value database for Go., `https://github.com/boltdb/bolt`. Accessed: 2022-01-28.

[4] : cockroachdb/cockroach: CockroachDB - the open source, cloud-native distributed SQL database., `https://github.com/cockroachdb/cockroach`. Accessed: 2022-01-28.

[5] : ctrlflow package, `https://golang.org/x/tools/go/analysis/passes/ctrlflow`. Accessed: 2021-12-21.

[6] : Data Race Detector - The Go Programming Language, `https://go.dev/doc/articles/race_detector`. Accessed: 2022-01-31.

[7] : etcd-io/etcd: Distributed reliable key-value store for the most critical data of a distributed system, `https://github.com/etcd-io/etcd`. Accessed: 2022-01-28.

[8] : fmt command - cmd/fmt - pkg.go.dev, `https://pkg.go.dev/cmd/gofmt@go1.17.6`. Accessed: 2022-01-14.

[9] : Frequently Asked Questions (FAQ), `https://go.dev/doc/faq`. Accessed: 2022-08-01.

[10] : GNU Time - GNU Project - Free Software Foundation, `https://www.gnu.org/software/time/`. Accessed: 2022-01-30.

[11] : google/gvisor: Application Kernel for Containers, `https://github.com/google/gvisor`. Accessed: 2022-01-30.

[12] : grpc/grpc-go: The Go language implementation of gRPC. HTTP/2 based RPC, `https://github.com/grpc/grpc-go`. Accessed: 2022-01-28.

[13] : Home — mCRL2 202106.0 documentation, `https://www.mcrl2.org/web/user_manual/index.html`. Accessed: 2022-01-31.

[14] : Iris Project, `https://iris-project.org/`. Accessed: 2022-01-31.

[15] : kubernetes/kubernetes: Production-Grade Container Scheduling and Management, `https://github.com/kubernetes/kubernetes`. Accessed: 2022-01-28.

[16] : moby/moby: Moby Project - a collaborative project for the container ecosystem to assemble container-based systems, `https://github.com/moby/moby`. Accessed: 2022-01-28.

[17] : Stream (Java Platform SE 8), `https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Stream.html`. Accessed: 2022-02-01.

[18] : vet command - cmd/vet - pkg.go.dev, `https://pkg.go.dev/cmd/vet@go1.17.6`. Accessed: 2022-01-14.

[19] : Merge pull request #7301 from vieux/-fix_goroutines_leak_exit_code · moby/moby@64579f5, `https://github.com/moby/moby/commit/64579f51fcb439c36377c0068ccc9a007b368b5a`, 8 2014. Accessed: 2022-01-30.

[20] : Merge pull request #17155 from janten/meitner · moby/moby@8eba018, `https://github.com/moby/moby/commit/8eba018b610e425960ffb4620dd7a0bfde5238d9`, 10 2015. Accessed: 2022-01-30.

[21] : Merge pull request #583 from kkaneda/kkaneda/use_has_space · cockroachdb/cockroach@a889f82, `https://github.com/cockroachdb/cockroach/commit/a889f828214fcfd76f7f2689b74cb4bec2bd23a1`, 4 2015. Accessed: 2022-01-30.

[22] : Merge pull request #20674 from caesarxuchao/decode-status · kubernetes/kubernetes@2e7993e, `https://github.com/kubernetes/kube`

rnetes/commit/2e7993e05708c484a69291c6ddee
f6f3bcc9ee23, 2 2016. Accessed: 2022-01-30.

[23] : Merge pull request #3652 from Bram-Gruneir/3270 · cockroachdb/cockroach@1f70b73, https://github.com/cockroachdb/cockroach/commit/1f70b73ae293d9cb18a0927e79ddc28d9e5cf696, 1 2016. Accessed: 2022-01-30.

[24] : Merge pull request #794 from smallfish/master · grpc/grpc-go@7c6103d, https://github.com/grpc/grpc-go/commit/7c6103d142da5fd11a36964c3a42cf698056b00b, 7 2016. Accessed: 2022-01-30.

[25] : Merge pull request #9924 from petermattis/pmattis/rm-resource-doc · cockroachdb/cockroach@d7bb465, https://github.com/cockroachdb/cockroach/commit/d7bb465e695c8d6e53d3e2d5664e764fe4f60f0d, 10 2016. Accessed: 2022-01-30.

[26] : Merge pull request #45179 from yguo0905/ubuntu-node-e2e-test · kubernetes/kubernetes @07424af, https://github.com/kubernetes/kubernetes/commit/07424af12b79fe214549e990e014d302b37df5c8, 5 2017. Accessed: 2022-01-30.

[27] : CVE-2022-31623, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31623, May 2022. Accessed: 2022-08-03.

[28] Allen, F. E.: Control flow analysis, *ACM Sigplan Notices*, Vol. 5, No. 7(1970), pp. 1–19.

[29] Bjørner, N. and de Moura, L.: Z3: An efficient SMT solver, *Tools and Algorithms for the Construction and Analysis of Systems,(TACAS' 08)*, (2008).

[30] Chajed, T., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N.: Verifying concurrent Go code in Coq with Goose, *The Sixth International Workshop on Coq for Programming Languages, CoqPL*, Vol. 2020, 2020.

[31] Dilley, N. and Lange, J.: An Empirical Study of Messaging Passing Concurrency in Go Projects.

[32] Dilley, N. and Lange, J.: Automated Verification of Go Programs via Bounded Model Checking, *International Conference on Automated Software Engineering (ASE). IEEE/ACM*, 2021.

[33] Gabet, J. and Yoshida, N.: Static race detection and mutex safety and liveness for go programs, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[34] Hoare, C. A. R.: Communicating sequential processes, *Communications of the ACM*, Vol. 21, No. 8(1978), pp. 666–677.

[35] Jin, G., Zhang, W., and Deng, D.: Automated concurrency-bug fixing, *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 221–236.

[36] Khatchadourian, R., Tang, Y., and Bagherzadeh, M.: Safe automated refactoring for intelligent parallelization of Java 8 streams, *Sci. Comput. Program.*, Vol. 195, No. 102476(2020), pp. 102476.

[37] Khoshnood, S., Kusano, M., and Wang, C.: ConcBugAssist: constraint solving for diagnosis and repair of concurrency bugs, *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, New York, NY, USA, Association for Computing Machinery, July 2015, pp. 165–176.

[38] Lange, J., Ng, N., Toninho, B., and Yoshida, N.: Fencing off Go: Liveness and safety for channel-based programming, *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, January 2017, pp. 748–761.

[39] Lange, J., Ng, N., Toninho, B., and Yoshida, N.: A Static Verification Framework for Message Passing in Go Using Behavioural Types, *Proceedings - International Conference on Software Engineering*, 2018.

[40] Lange, J. and Yoshida, N.: Verifying Asynchronous Interactions via Communicating Session Automata, *Computer Aided Verification*, Dillig, I. and Tasiran, S.(eds.), Cham, Springer International Publishing, 2019, pp. 97–117.

[41] Liu, Z., Zhu, S., Qin, B., Chen, H., and Song, L.: Automatically detecting and fixing concurrency bugs in go software systems, *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 616–629.

[42] McCool, M. D., Robison, A. D., Reinders, J., 清文菅原, and エクセルソフト株式会社: 構造化並列プログラミング：効率良い計算を行うためのパターン, カットシステム, 2013.

[43] Ng, N. and Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis, *Proceedings of CC 2016: The 25th International Conference on Compiler Construction*, Association for Computing Machinery, Inc, March 2016, pp. 174–184.

[44] Ng, N. and Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis, *Proceedings of CC 2016: The 25th International Conference on Compiler Construction*, Association for Computing Machinery, Inc, March 2016, pp. 174–184.

[45] Stadtmüller, K., Sulzmann, M., and Thiemann, P.: Static trace-based deadlock analysis for synchronous mini-go, *Programming Languages and Systems*, Lecture notes in computer science, Springer International Publishing, Cham, 2016, pp. 116–136.

[46] Suzuki, D.: Go を使いこなせる組織作りの取り組み / DeNA.go ＃ 1, https://speakerdeck.com/daisuzu/dena-dot-go-number-1?slide=10, 5 2019. Accessed: 2022-01-03.

[47] Tarjan, R.: Depth-first search and linear graph

algorithms, *SIAM journal on computing*, Vol. 1, No. 2(1972), pp. 146–160.

[48] Tu, T., Liu, X., Song, L., and Zhang, Y.: Understanding real-world concurrency bugs in Go, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 865–878.

[49] Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., and Wei, J.: A comprehensive study on real world concurrency bugs in Node.js, *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, October 2017, pp. 520–531.

[50] Wei, T., Mao, J., Zou, W., and Chen, Y.: A new algorithm for identifying loops in decompilation, *International Static Analysis Symposium*, Springer, 2007, pp. 170–183.