Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems

Ziheng Liu
Pennsylvania State University
USA

Shuofei Zhu
Pennsylvania State University
USA

Boqin Qin
Beijing Univ. of Posts and Telecom.
China

Hao Chen
University of California, Davis
USA

Linhai Song
Pennsylvania State University
USA

ABSTRACT

Go is a statically typed programming language designed for efficient and reliable concurrent programming. For this purpose, Go provides lightweight goroutines and recommends passing messages using channels as a less error-prone means of thread communication. Go has become increasingly popular in recent years and has been adopted to build many important infrastructure software systems. However, a recent empirical study shows that concurrency bugs, especially those due to misuse of channels, exist widely in Go. These bugs severely hurt the reliability of Go concurrent systems.

To fight Go concurrency bugs caused by misuse of channels, this paper proposes a static concurrency bug detection system, GCatch, and an automated concurrency bug fixing system, GFix. After disentangling an input Go program, GCatch models the complex channel operations in Go using a novel constraint system and applies a constraint solver to identify blocking bugs. GFix automatically patches blocking bugs detected by GCatch using Go’s channel-related language features. We apply GCatch and GFix to 21 popular Go applications, including Docker, Kubernetes, and gRPC. In total, GCatch finds 149 previously unknown blocking bugs due to misuse of channels and GFix successfully fixes 124 of them. We have reported all detected bugs and generated patches to developers. So far, developers have fixed 125 blocking misuse-of-channel bugs based on our reporting. Among them, 87 bugs are fixed by applying GFix’s patches directly.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; Software reliability.

KEYWORDS

Go; Concurrency Bugs; Bug Detection; Bug Fixing; Static Analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS ’21, April 19–23, 2021, Virtual, MI, USA.
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8317-2/21/04...$15.00
https://doi.org/10.1145/3445814.3446756

1 INTRODUCTION

Go is a statically typed programming language designed by Google in 2009 [13]. In recent years, Go has gained increasing popularity in building software in production environments. These Go programs range from libraries [5] and command-line tools [1, 6] to systems software, including container systems [12, 22], databases [3, 8], and blockchain systems [16].

One major design goal of Go is to provide an efficient and safe way for developers to write concurrent programs [14]. To achieve this purpose, Go provides easily created lightweight threads (called goroutines); it also advocates the use of channels to explicitly pass messages across goroutines, on the assumption that message-passing concurrency is less error-prone than shared-memory concurrency supported by traditional programming languages [24, 31, 69]. In addition, Go also provides several unique primitives and libraries for concurrent programming.

Unfortunately, Go programs still contain many concurrency bugs [77], the type of bugs that are most difficult to debug [30, 53] and severely hurt the reliability of multi-threaded software systems [15, 78]. Moreover, a recent empirical study [87] shows that message passing is just as error-prone as shared memory, and that misuse of channels is even more likely to cause blocking bugs (e.g., deadlock) than misuse of mutexes.

A previously unknown concurrency bug in Docker is shown in Figure 1. Function Exec() creates a child goroutine at line 5 to duplicate the content of a .Reader. After the duplication, the child goroutine sends err to the parent goroutine through channel outDone to notify the parent about completion and any possible error (line 7). Since outDone is an unbuffered channel (line 3), the child blocks at line 7 until the parent receives from outDone. Meanwhile, the parent blocks at the select at line 9 until it either receives err from the child (line 10) or receives a message from ctx.Done() (line 13), indicating the entire task can be halted. If the message from ctx.Done() arrives earlier, or if the two messages arrive concurrently and Go’s runtime non-deterministically chooses the second case to execute, the parent will return from function Exec(). No other goroutine can pull messages from outDone, leaving the child goroutine permanently blocked at line 7.
This bug demonstrates the complexity of Go’s concurrency features. Programmers have to have a good understanding of when a channel operation blocks and how select waits for multiple channel operations. Otherwise, it is easy for them to make similar mistakes when programming Go. Since Go is being widely adopted, it is increasingly urgent to fight concurrency bugs in Go, especially those caused by misuse of channels, since Go advocates using channels for thread communication and many developers choose Go because of its good support for channels [72, 86].

However, existing techniques cannot effectively detect channel-related concurrency bugs in large Go software systems. First, concurrency bug detection techniques designed for classic programming languages [49, 58, 67, 68, 75, 82] mainly focus on analyzing shared-memory accesses or shared-memory primitives. Thus, they cannot detect bugs caused by misuse of channels in Go. Second, since message-passing operations in MPI are different from those in Go, techniques designed for detecting deadlocks in MPI programs [38, 41, 42, 83, 88] cannot be applied to Go programs. Third, the three concurrency bug detectors released by the Go team [9, 11, 19] cover only limited buggy code patterns and cannot identify the majority of Go concurrency bugs in the real world [87]. Fourth, although recent techniques can use model checking to identify blocking bugs in Go [39, 59, 60, 70, 80], those techniques analyze each input program and all its synchronization primitives as a whole. Due to the exponential complexity of model checking, those techniques can handle only small programs with a few primitives, and cannot scale to large systems software containing millions of lines of code and hundreds of primitives (e.g., Docker, Kubernetes).

In this paper, we propose GCatch, a static detection system that can effectively and accurately identify concurrency bugs in large Go systems software. As shown in Figure 2, GCatch focuses on blocking misuse-of-channel (BMOC) bugs ("BMOC Detector" in Figure 2), since the majority of channel-related concurrency bugs in Go are blocking bugs [87]. GCatch also contains five additional detectors ("Traditional Detectors" in Figure 2) that rely on ideas effective at discovering concurrency bugs in traditional programming languages.

The design of GCatch comprises two steps. To scale to large Go software, GCatch conducts reachability analysis to compute the relationship between synchronization primitives of an input program, and leverages that relationship to disentangle the primitives into small groups. GCatch inspects each group only in a small program scope. To detect BMOC bugs, GCatch enumerates execution paths for all goroutines executed in a small program scope, uses constraints to precisely describe how synchronization operations proceed and block, and invokes Z3 [33] to search for possible executions that would lead some synchronization operations to block forever (thereby detecting blocking bugs).

The key challenge of building GCatch lies in modeling channel operations using constraints. Since a channel’s behavior depends on its states (e.g., the number of elements in the channel, closed or not), channel operations are much more complex to model than the synchronization operations (e.g., locking/unlocking) already covered in existing constraint systems [45, 48, 57, 90]. We model channel operations by associating each channel with state variables and defining how to update state variables when a channel operation proceeds. Our constraint system is the first to consider states of synchronization primitives. It is very different from existing constraint systems and models used in the previous model checking techniques [39, 59, 60, 70, 80].

A BMOC bug will continue to hurt the system’s reliability until it is fixed. Thus, we further design an automated concurrency bug fixing system (GFix) to patch BMOC bugs detected by GCatch (Figure 2). GFix first conducts static analysis to categorize input BMOC bugs into three groups. It then automatically increases channel buffer sizes or uses keyword “defer” or “select” to change blocking channel operations to be non-blocking and fix bugs in each group. One challenge of automated bug fixing is generating readable patches that will be more readily accepted by developers. GFix synthesizes patches using Go’s channel-related language features, which are powerful and already frequently used by programmers. Thus, GFix’s patches only change a few lines of code and align with programmers’ usual practice. They are easily validated and accepted by developers.

The bug and its patch shown in Figure 1 confirm the effectiveness of GCatch and GFix. After inspecting Docker, GCatch identifies the previously unknown bug by reporting that when the parent goroutine chooses the second case, the child goroutine endless blocks at line 7, which is the correct root cause of the bug. GFix changes one line of code to increase outDone’s buffer size from zero to one, which successfully fixes the bug. We submitted the bug and the patch to Docker developers. They directly applied our patch in a more recent Docker version.

We evaluate GCatch and GFix on 21 popular real-world Go software systems including Docker, Kubernetes, and gRPC. In total, GCatch finds 149 previously unknown BMOC bugs and reports 51 false positives. We reported all detected bugs to developers. So far, 125 BMOC bugs have been fixed based on our reporting. The largest application used in our evaluation (Kubernetes) contains more than three million lines of code. GCatch can finish examining it in 25.6

```
1 func Exec(ctx context.Context, ...) (ExResult, error) {
2   var oBuf, eBuf bytes.Buffer
3   outDone := make(chan error)
4   go func() {
5       _, err = StdCopy(&oBuf, &eBuf, a.Reader)
6       outDone <- err // block
7   }()
8 }
9 select {
10   case err := <-outDone:
11       if err != nil {
12           return ExecResult{}, err
13       }
14       return ExecResult{}, ctx.Err()
15   }
16   return ExecResult{oBuf: &oBuf, eBuf: &eBuf}, nil
17 }
```

Figure 1: A previously unknown Docker bug and its patch.
hours and finds 15 BMOC bugs, demonstrating its capability to analyze large Go software. GFix generates patches for 124 detected BMOC bugs. All the patches are correct. On average, each patch incurs less than 0.26% runtime overhead and changes 2.67 lines of code. Eighty-seven of the generated patches have been applied by developers directly. Overall, GCatch can effectively detect BMOC bugs in large, real Go software, and GFix can synthesize correct patches with good performance and readability for detected bugs. GCatch and GFix constitute an end-to-end system to combat BMOC bugs and improve the reliability of Go software systems.

In summary, we make the following contributions:

- Based on a disentangling policy and a novel constraint system, we build an effective concurrency bug detection system that can analyze large Go systems software.
- We design an automated bug fixing system for BMOC bugs in Go. The system contains three fixing strategies relying on Go’s channel-related language features. The system generates correct patches with good performance and readability.
- We conduct thorough experiments to evaluate our systems. We identify and fix hundreds of previously unknown concurrency bugs in real Go software.

All our code and experimental data (including detected bugs and generated patches) can be found at https://github.com/system-pclub/GCatch.

2 BACKGROUND

This section gives some background of this project, including Go’s synchronization primitives, concurrency bugs found in Go, and limitations of existing constraint systems.

2.1 Concurrency in Go

Go supports concurrency primitives for both passing messages between goroutines and protecting shared memory accesses.

Message Passing. Channel (chan) is the most commonly used message-passing primitive in Go [87]. It can send data across goroutines and synchronize different goroutines to implement complex functionalities [25, 73]. Go supports two types of channels (unbuffered and buffered) and three types of channel operations (receiving, sending, and closing). Whether a channel operation blocks and what its return value is depend on the channel’s states (e.g., full or not, closed or not). For example, if a channel’s buffer is full, a goroutine sending data to the channel will block until another goroutine receives data from that channel or closes it, while the goroutine that sends data will not block, if the channel’s buffer still has empty slots. If a channel is closed, a goroutine seeking to receive data from the channel will immediately receive a zero value (e.g., ”” for string).

Go’s select statement (e.g., line 9 in Figure 1) allows a goroutine to wait for multiple channel operations. A goroutine blocks at a select until one of the select’s channel operations can proceed, unless the select has a default clause. If multiple channel operations can proceed at the same time, Go’s runtime will non-deterministically choose one to execute.

Protecting Shared Memory Accesses. Go allows multiple goroutines to access the same memory, and also provides several primitives to protect shared memory accesses, including lock (RWMutex), read/write lock (RWLock), condition variable (Cond), atomic instructions (atomic), and a primitive to wait for multiple goroutines to finish their tasks (WaitGroup).

2.2 Concurrency Bugs in Go

Unfortunately, concurrency bugs are more likely to happen in Go [77]. Thus, we need to detect them and fix them. A previous empirical study [87] categorized Go concurrency bugs into two groups: blocking bugs, where one or more goroutines are unintentionally stuck in their execution (e.g., deadlock), and non-blocking bugs, where all goroutines can finish their execution but with undesired results (e.g., data race). The study further divided bugs in each category into several subcategories based on which primitive causes a bug. The proportion of bugs in each (sub)category motivates the design of GCatch and GFix. Since we want to combat channel-related concurrency bugs and the study reported that most bugs due to misuse of channels are blocking bugs [87], we concentrate our efforts on combating blocking misuse-of-channel (BMOC) bugs. According to the study, BMOC bugs are usually caused by errors when only using channels or using channels together with mutexes.

2.3 Existing Constraint Systems

Previously, constraint solving was used to dynamically detect concurrency bugs [45, 57, 90]. Different types of constraints were defined to describe how a concurrent system works from different aspects, including synchronization enforced by locking/unlocking operations (lock constraints), instruction orders enforced by thread creation or wait/signal (partial-order constraints), instruction orders caused by program flow (memory order constraints), and the occurrence of bugs like data races and atomicity violations (bug constraints).

Unfortunately, existing constraint systems don’t consider the state of synchronization primitives. For example, the existing constraint systems model the mutual exclusion supported by a mutex as a requirement that the critical sections protected by the mutex not execute concurrently, while not modeling the mutual exclusion by considering whether the mutex is locked or not. As discussed in Section 2.1, channel operations are so complex that they can only be described precisely after taking account of channels’ states (e.g., the number of elements in a channel, closed or not). It is neither straightforward nor trivial to extend existing constraint systems to apply to channels, since there are at least two challenges to solve: 1) how to model a channel’s states? and 2) how to update a channel’s states after an operation is taken on the channel? We will present our solutions in Section 3.4.

3 DETECTING GO CONCURRENCY BUGS

This section discusses GCatch in detail. As shown in Figure 2, GCatch takes Go source code as input and reports detected bugs for GFix or developers to fix.

The key benefit of GCatch is its capability to detect BMOC bugs in large Go software systems (“BMOC Detector” in Figure 2). To achieve this purpose, GCatch’s design mainly contains two components: an effective disentangling policy to separate synchronization primitives of an input program and a novel constraint system to
Algorithm 1 BMOC Bug Detection

Require: An input Go program (P)
1: function (P)
2: \( C_{\text{Graph}} \leftarrow \text{BuildCallGraph}(P) \)
3: \( \text{Alias} \leftarrow \text{ConductAliasAnalysis}(C_{\text{Graph}}) \)
4: \( \text{Primitives} \leftarrow \text{SearchSynPrimitives}(C_{\text{Graph}}, \text{Alias}) \)
5: \( OP_{\text{map}} \leftarrow \text{SearchSynOperations}(\text{Primitives}, C_{\text{Graph}}, \text{Alias}) \)
6: \( /^{*} \text{more details of lines 7 and 8 are in Section 3.2} /^{*} \)
7: \( D_{\text{Graph}} \leftarrow \text{BuildDependGraph}(OP_{\text{map}}, C_{\text{Graph}}) \)
8: \( \text{for each channel } c \text{ in } \text{Primitives} \{ \)
9: \( \text{scope}, P_{\text{set}} \leftarrow \text{DisentanglingAnalysis}(c, OP_{\text{map}}, D_{\text{Graph}}) \)
10: \( \text{GOset} \leftarrow \text{SearchGoroutines}(c, \text{scope}) \)
11: \( /^{*} \text{more details of line 12 are in Section 3.3} /^{*} \)
12: \( PC_{\text{s}} \leftarrow \text{ComputePathCombination}(\text{GOset}, P_{\text{set}}, OP_{\text{map}}, \text{scope}) \)
13: \( \text{for each path combination } pc \text{ in } PC_{\text{s}} \{ \)
14: \( /^{*} \text{more details of lines 15 - 25 are in Section 3.4} /^{*} \)
15: \( \text{Groups} \leftarrow \text{ComputeSuspiciousOpGroups}(pc, P_{\text{set}}, \text{GOset}) \)
16: \( \text{for each group } (g) \text{ in } \text{Groups} \{ \)
17: \( \Phi_{B} \leftarrow \text{ReachConstraint}(g, P_{\text{set}}) \)
18: \( \Phi_{S} \leftarrow \text{BlockConstraint}(g) \)
19: \( \Phi \leftarrow \Phi_{B} \land \Phi_{S} \)
20: \( \text{if } Z3 \text{ finds a solution for } \Phi \text{ then} \)
21: \( \text{ReportBug}(g, s) \)
22: \( \text{end if} \)
23: \( \text{end for} \)
24: \( \text{end for} \)
25: \( \text{end for} \)
26: \( \text{end function} \)

model channel-related concurrency features in Go. We will mainly present the BMOC detector in this section.

Besides the BMOC detector, GCatch also contains five traditional detectors for concurrency bugs that also appear in classic programming languages (“Traditional Detectors” in Figure 2). We will briefly describe them at the end of this section.

3.1 Workflow
To detect BMOC bugs, GCatch takes several steps as sketched out in Algorithm 1. We explain the steps as follows.

First, GCatch inspects the whole input program to search for concurrency primitives and their operations (lines 2–5 in Algorithm 1). GCatch distinguishes primitives using their static creation sites and leverages alias analysis to determine whether an operation is performed on a primitive. For example, GCatch uses line 3 to represent channel outDone in Figure 1 and finds three operations (i.e., lines 3, 7, 10) for it.

Second, after disentangling an input program, GCatch iterates all its channels (lines 8–25 in Algorithm 1). Given a channel c, GCatch only inspects c and a few other related primitives (\( P_{\text{set}} \)) in a small analysis scope (\( \text{scope} \)). We will discuss the disentangling policy in Section 3.2.

Third, GCatch determines the goroutine set (\( \text{GOset} \)) that accesses channel c by inspecting all goroutines created in c’s analysis scope (line 10 in Algorithm 1). For the bug in Figure 1, the analysis scope of channel outDone is identified as extending from its creation site (line 3) to the end of function Exec(). Besides the parent goroutine, GCatch also identifies the child goroutine created at line 5 as accessing channel outDone.

Fourth, GCatch computes path combinations by enumerating all possible execution paths for each goroutine in \( \text{GOset} \) (line 12 in Algorithm 1). GCatch provides heuristics to filter out combinations that are either infeasible or unlikely to contain BMOC bugs. We will discuss more details in Section 3.3.

Fifth, GCatch computes all suspicious groups for a path combination at line 15. Each suspicious group contains synchronization operations that together can potentially block forever.

Sixth, GCatch uses a constraint solver Z3 [33] to examine all suspicious groups (lines 16–23 in Algorithm 1). For each group, GCatch computes the constraints (\( \Phi_{B} \)) for the program to execute just before the group operations and the constraints (\( \Phi_{S} \)) for all group operations to block. If Z3 finds a solution for the conjunction of \( \Phi_{B} \) and \( \Phi_{S} \) at line 20, GCatch detects a bug. We will discuss how to compute suspicious groups and how our constraint system works in Section 3.4.

3.2 Disentangling Input Software
If GCatch were to analyze the whole input program and all its primitives, it would be difficult to scale to large Go software systems. Thus, GCatch disentangles each input program and inspects each channel separately to determine whether that channel causes any BMOC bugs (line 8–25 in Algorithm 1).

Given that any one channel c is unlikely to be used throughout the whole input program, GCatch only needs to analyze c in its usage scope (\( \text{scope} \)) at line 9. Moreover, if GCatch only examines c without considering other related primitives, GCatch cannot detect bugs arising when using c and other primitives together. Thus, GCatch needs to compute which primitives (\( P_{\text{set}} \)) at line 9) must be analyzed together with c.

How to compute \( \text{scope} \)? GCatch first analyzes the call graph of the input program and searches for the lowest common ancestor (LCA) function that can invoke all operations of c directly or indirectly. Then, GCatch defines \( \text{scope} \) as extending from c’s creation site to the end of the LCA function, including all functions called directly and indirectly in between. When analyzing a library, GCatch may identify a set of functions whose combination can cover all operations of c. In this case, GCatch computes a scope for each of the identified functions and consider scope of c as the union of the computed scopes.

How to compute \( P_{\text{set}} \)? GCatch builds a graph (\( D_{\text{Graph}} \)) at line 7 in Algorithm 1 by computing the dependence relationship for each pair of synchronization primitives. Later, at line 9, GCatch queries the graph to get \( P_{\text{set}} \) for channel c.

Given primitive a and primitive b, GCatch computes their dependence by checking whether they are in one of the following scenarios: 1) if one of a’s operations with the capability to unblock another operation of a (e.g., sending, unlocking) is reachable from one of b’s operations that can block (e.g., receiving, locking), then a depends on b, since whether a’s blocking operation can proceed depends on how b’s blocking operation executes; 2) if both a and b are channels and a select waits for operations of a and b, then a and b depend on each other. Since a select can only choose one channel operation to process, whether an operation in the select can proceed depends on whether other operations in the same select cannot make progress.

We also consider dependence to be transitive, which means if a depends on b and b depends on c, then a depends on c.

GCatch computes \( P_{\text{set}} \) of channel c as containing c and all other synchronization primitives that have a smaller scope and a circular dependency relationship with it.
Running Example. For channel outDone in Figure 1, its \( P_{\text{set}} \) only contains itself. Although channel ctx.Done() and channel outDone circularly depend on each other (in the same select), ctx.Done() has a larger scope than outDone, so that it is not in outDone’s \( P_{\text{set}} \) and it is ignored by GCatch when GCatch analyzes outDone. The two channels will be inspected together when GCatch analyzes ctx.Done().

3.3 Computing Path Combinations

For each goroutine in \( G_{\text{set}} \), GCatch conducts an inter-procedural, depth-first search to enumerate all its execution paths within scope (line 12 in Algorithm 1). GCatch processes instruction by instruction. Given a call instruction, if the callee does not execute any operation of any primitive in \( P_{\text{set}} \) directly or indirectly, GCatch speeds up its analysis by ignoring the function call. Given a loop whose iteration number cannot be computed statically, GCatch iterates the loop at most twice to avoid path explosion, which may lead to both false positives and false negatives, since GCatch may mistakenly count the number of sending or receiving operations inside the loop.

GCatch enumerates all possible path combinations for all goroutines in \( G_{\text{set}} \). GCatch filters out combinations that either include an infeasible execution path or do not contain any blocking operation. For example, GCatch inspects branch conditions only involving read-only variables and constants and filters out path combinations containing conflicting branch conditions. As another example, GCatch analyzes the terminating conditions for all loops in each combination. If GCatch identifies a combination involving two loops with the same loop terminating condition but different loop iteration numbers, then GCatch filters out that combination.

Running Example. For the bug in Figure 1, function StdCopy() at line 6 does not contain any synchronization operation. Thus, GCatch ignores it when enumerating paths for the child goroutine. GCatch only finds one path for the child. Since there is a select statement with two cases at line 9 and an if statement at line 11, GCatch finds three possible paths for the parent goroutine. In total, GCatch identifies three path combinations.

3.4 Identifying BMOC Bugs

Detecting a BMOC bug is equivalent to identifying a group of operations that together can block forever. GCatch takes two steps to achieve this purpose: 1) it identifies suspicious groups (line 15 in Algorithm 1) with synchronization operations belonging to primitives in \( P_{\text{set}} \), from different goroutines, and unable to unblock other operations in the same group; and 2) it leverages Z3 to validate each group by checking whether all instructions before the group operations can execute (\( \Phi_{\text{R}} \)) and whether all group operations can block (\( \Phi_{s} \)) (lines 16–23).

The current version of GCatch only models channel operations and mutex operations, since they cause most BMOC bugs [87]. GCatch changes every mutex to a channel, so that we will mainly focus our explanation on modeling channels.

How to model channel states? As discussed in Section 2.3, the challenges of using constraints to describe channel operations lie in how to model and update channel states. There are two types of states that control whether a channel operation blocks: 1) how many elements are in the channel; and 2) whether the channel is closed. We model them as follows.

To model how many elements are in a channel and whether a channel is full, we associate each channel with a \( BS \) constant denoting the channel buffer size, and associate each sending/receiving operation with a \( CB \) variable denoting the number of elements in the channel before conducting the operation. For example, \( CB_{7} \) indicates that before the sending operation at line 7 in Figure 1, \( CB_{7} \) elements are in channel outDone. The value of a \( CB \) variable is computed as the number of sending operations minus the number of receiving operations, where the sending and receiving operations are conducted on the same channel and before the operation of the \( CB \) variable.

To model whether a channel is closed, we associate each receiving operation with a binary \( CLOSE \) variable and compute the variable value by checking whether a closing operation is conducted earlier on the same channel.

How to compute \( \Phi_{R} \)? \( \Phi_{R} \) represents all the necessary constraints for goroutines in \( G_{\text{set}} \) to execute just before operations in a suspicious group. Besides variables for channel states, \( \Phi_{s} \) contains another two types of variables. First, we associate each instruction with an \( O \) variable denoting its execution order, e.g., \( O_{7} \) representing the order of the sending operation at line 7 in Figure 1. Second, we associate each pair of sending and receiving operations of the same channel but in different goroutines with a binary \( P \) variable denoting whether the two operations match and unblock each other. If \( P_{(s_{i}, r_{j})} = 1 \), then the sending operation at line \( i \) and the receiving operation at line \( j \) execute at the same time (\( O_{s_{i}} = O_{r_{j}} \)).

\( \Phi_{R} \) is constructed by a conjunction of three sub-formulae: 1) \( \Phi_{\text{order}} \) denotes the instruction orders enforced by each execution path; 2) \( \Phi_{\text{spawn}} \) denotes the orders enforced by goroutine creations; and 3) \( \Phi_{\text{sync}} \) denotes the constraints required by the proceeding of synchronization operations. When computing \( \Phi_{\text{sync}} \), GCatch only considers synchronization operations belonging to primitives within \( P_{\text{set}} \). Since \( \Phi_{\text{order}} \) and \( \Phi_{\text{spawn}} \) were discussed in previous literature [48, 90], we only explain how to compute \( \Phi_{\text{sync}} \).

Modeling channels. GCatch models sending operations, receiving operations, and select statements as follows.

If a sending operation at line \( i (s_{i}) \) proceeds, one of the following two conditions must be satisfied: 1) the channel buffer is not full, or 2) one and only one receiving operation matches the sending. We use \( R \) to denote all receiving operations executed by different goroutines on the same channel. The computed constraints for \( s_{i} \) are written as follows:

\[
CB_{i} < BS \lor \left[ \bigvee_{r \in R} P_{(s_{i}, r)} = 1 \land O_{s_{i}} = O_{r} \land \sum_{r \in R} P_{(s_{i}, r)} = 1 \right]
\]

Similarly, the proceeding of a receiving operation at line \( j (r_{j}) \) must satisfy one of the following conditions: 1) at least one element is in the channel; 2) the channel is closed; or 3) one and only one sending matches the receiving. We use \( S \) to denote all sending operations executed by different goroutines on the same channel.
The constraints for $r_j$ are written as follows:

$$C_{Bj} > 0 \lor C_{Ej} \lor \left[ \bigvee_{s \in S} P_{(s,r_j)} = 1 \land O_{s\alpha} = O_{r_j} \land \sum_{s \in S} P_{(s,r_j)} = 1 \right]$$

When a select proceeds, there are two possible cases. First, if the select chooses its default clause, all the channel operations in that select block, and GCatch computes the constraints for all the channel operations unable to proceed. Second, if the select chooses a channel operation, GCatch computes the constraints for the operation to make progress.

**Modeling mutexes.** GCatch changes each mutex to a channel with buffer size one, changes its locking operations to sending to the channel, and changes its unlocking operations to receiving from the channel. Then, GCatch can compute constraints for mutex operations in the same way as channel operations.

**How to compute $\Phi_B$?** Similarly, $\Phi_B$ for a suspicious group is constructed by a conjunction of two sub-formulae: 1) $\Phi_{\text{order}}$ requires that instructions before the group operations must have smaller $O$ variable values, compared with the group operations, and 2) $\Phi_{\text{sync}}$ requires each group operation to be unable to make progress.

A receiving operation blocks when there is no element in the channel. The channel is not closed, and no sending operation is paired with the receiving. A sending operation blocks when the channel is full and there is no paired receiving. A select blocks when the select does not have a default clause and none of its channel operations can proceed.

**Working Example.** When analyzing channel outDone in Figure 1, we assume the path combination to be processed includes path 3-5-9-13-14 in the parent goroutine and path 6-7 in the child goroutine. Since ctx.Done() is not in outDone's $P_{\text{out}}$, only the sending operation at line 7 satisfies the requirements for being in a suspicious group. Thus, GCatch only finds one suspicious group containing the sending at line 7.

For $\Phi_R$, $\Phi_{\text{order}}$ is $O_3 < ... < O_{13} < O_{14}$ and $O_6 < O_7$, which is enforced by the two paths in the path combination. $\Phi_{\text{spawn}}$ is “$O_5 < O_6$” required by the goroutine creation at line 5, and $\Phi_{\text{sync}}$ is empty, since there is no synchronization locking to any primitive in the $P_{\text{out}}$ on the path of the parent goroutine and on the path of the child goroutine before line 7.

For $\Phi_B$, $\Phi_{\text{order}}$ is $O_3 < O_7 < ... < O_{14} < O_7$ and $O_6 < O_7$, which means instructions before the instruction in the suspicious group (the sending at line 7) execute earlier. $\Phi_{\text{sync}}$ is “$CB_{Bj} = BS$”, representing channel outDone is full. Channel outDone is an unbuffered channel (line 3), and thus BS is 0.

Z3 successfully finds a solution for the conjunction of $\Phi_R$ and $\Phi_B$. The solution is $O_3 = 0 \land ... \land O_{14} = 4 \land O_6 = 5 \land O_7 = 6 \land CB_{Bj} = 0$, which means when the program executes in the order $3 \rightarrow ... \rightarrow 14 \rightarrow 6 \rightarrow 7$, the child blocks at sending operation at line 7 forever.

### 4.1 Overview

The current version of GFix can only fix a subset of BMOC bugs, specifically BMOC bugs involving two goroutines and a single channel. We leave more complex BMOC bugs (e.g., those with more channels) to future work.

We formalize the problem scope of GFix as follows. Given a BMOC bug in the scope, we assume the two involved goroutines are Go-A and Go-B, and they interact with each other using local channel $c$. (Since we require that only Go-A and Go-B access $c$ and it is difficult to statically identify how many goroutines share a global channel, we require $c$ to be a local channel.) When the bug is triggered, Go-A fails to conduct an operation $o_1$ on $c$, causing Go-B to be blocked at another operation $o_2$ on $c$ forever. After fixing the bug, Go-B can continue its execution after $o_2$ or just stop its execution at $o_2$.

To guarantee a patch's correctness, GFix must make sure there is no race condition between instructions before $o_1$ in Go-A and instructions after $o_2$ in Go-B, if Go-B continues its execution, since the patch removes the order enforced by $o_1$ and $o_2$. GFix also needs to guarantee that the patch (i.e., forcing B to continue or stop) does
not change the original program semantics. For both of the two requirements, GFix needs to inspect instructions after o2 in Go-B; therefore we further limit GFix to only fix bugs where Go-B is a child goroutine created by Go-A to have a clear scope and context to analyze Go-B. We will use the parent (child) goroutine and Go-A (Go-B) interchangeably in this section.

GFix provides three fixing strategies based on three different directions for resolving bugs in the problem scope: how to make o2 non-blocking, how to force Go-A to always execute o1, and how to make Go-A notify Go-B of its problem and stop Go-B’s execution at o2. GFix does not take a generic fixing strategy, because it is difficult to automatically generate generic patches with good readability.

Next, we will discuss how GFix identifies bugs for each fixing strategy and the corresponding code transformation.

4.2 Strategy-I: Increasing Buffer Size

This strategy makes o2 non-blocking, so that Go-B can continue its computation after o2. If o2 is a receiving operation, we have to synthesize the received value for the computation after o2. This is difficult, if not impossible. However, if o2 is a sending operation, we can make o2 non-blocking simply by increasing channel c’s buffer size. If Go-B conducts multiple sending operations on channel c, increasing the buffer size may make sending operations other than o2 non-blocking as well, potentially violating the original synchronization requirement. Thus, this strategy fixes bugs where Go-B conducts one single sending operation on an unbuffered channel. We call these bugs single-sending bugs. To fix such bugs, GFix increases channel c’s buffer size from zero to one.

Although single-sending bugs sound specific, they reflect a common design pattern [85]. Go developers usually create a goroutine for a task with a message at the end to notify that the task is complete or to send out the result. For example, we find 125 goroutines are used in this way in Docker and Kubernetes. The bug in Figure 1 is a single-sending bug in Docker. Based on our formalization, Go-A is the parent goroutine executing function Exec, Go-B is the child goroutine created at line 5, and channel outDone is the buggy channel. Go-B conducts only one sending operation on channel outDone. The bug is fixed by increasing outDone’s buffer size at line 4 in Figure 1.

How to identify single-sending bugs? GCatch reports the blocking operations (e.g., line 7 in Figure 1) for each detected BMOC bug. GFix takes the blocking operations as input and uses the following four steps to decide whether the bug can be fixed by increasing the buffer size from zero to one.

First, GFix checks whether there is only one blocking operation reported, whether the operation is a sending operation, and whether the operation is on an unbuffered channel. If so, the channel is c and the operation is o2 in our formalization.

Second, GFix inspects whether channel c is shared (or accessed) by only two goroutines. The parent goroutine creating channel c is one goroutine that accesses c (e.g., the parent in Figure 1). GFix examines all child goroutines created in the variable scope of channel c to search for another goroutine accessing c. If there is more than one child goroutine accessing c, then the bug is not a single-sending bug. After identifying the child goroutine, GFix checks whether it is the one executing o2.

Figure 3: A missing-interaction bug in etcd.

Third, GFix conducts inter-procedural, path-sensitive analysis to count how many possible channel operations Go-B can conduct on c in one of its executions, and filters out cases where Go-B conducts operations on c other than o2 and cases where Go-B conducts o2 multiple times.

Fourth, GFix examines whether unblocking o2 violates the correctness and the original semantics. GFix inter-procedurally checks whether any instruction after o2 calls a library function, conducts a concurrency operation, or updates a variable defined outside Go-B. If so, GFix considers that its fix may cause a side effect beyond Go-B and that unblocking o2 may therefore cause a problem. In such a case, GFix does not fix the bug. For example, in Figure 1, the child goroutine does not execute any instruction after line 7 except for the implicit return. Thus, increasing outDone’s buffer size can unblock line 7 without changing the original program semantics.

4.3 Strategy-II: Deferring Channel Operation

This strategy forces o1 to be always executed, so that Go-B can be unblocked at o2. Since c is a local channel, when Go-A leaves c’s variable scope, it will not conduct any operation on c. If c’s scope ends at the end of a function, we can use keyword defer to guarantee that o1 will always be executed by Go-A, since Go’s runtime automatically executes all deferred operations in a function when the function returns.

We call bugs that can be fixed by this strategy missing-interaction bugs. They are triggered when Go-A leaves the function where c is valid (due to return or panic) without executing o1. To fix these bugs, we add a defer with o1 as its operand early enough to cover all execution paths of the function. We also remove the original o1s if they exist. Figure 3 shows an example. When t.Fatal() at line 9 executes, the testing function terminates and the sending at line 11 is skipped, causing the child created at line 6 to be blocked at line 16 forever. The patch defers the sending operation at lines 3 – 5 and also removes the original sending at line 11.

How to identify and fix missing-interaction bugs? GFix repeats the four steps outlined in Section 4.2 to identify possible cases. The only difference is that GFix allows o2 to be a receiving operation (e.g., line 16 in Figure 3). GFix then takes four extra steps for missing-interaction bugs.

First, GFix conducts inter-procedural analysis for all functions called by the function declaring channel c (e.g., TestRWDialer() in Figure 3) to examine whether any of them can cause a panic. If so, GFix treats the call sites (e.g., line 9 in Figure 3) in the same way as return in later steps.

Second, GFix checks whether every return in the function declaring c is dominated by a static o1 instruction. There can be multiple
Figure 4: A multiple-operations bug in Go-Ethereum.

static o1s, but when the parent goroutine takes a path without any static o1 on it, o2 cannot be unblocked. For example, when the parent in Figure 3 executes line 9 and leaves TestRWDialer(), the BMOC bug is triggered.

Since the patch basically moves an existing o1 to a point just before the return post-dominating it, GFix needs to check whether moving the o1 is safe. In the third step, GFix inspects each existing o1 by examining whether instructions between the o1 and the return post-dominating it contain any synchronization operation or have any data dependence relationship with o1 (if o1 is a receiving operation). If so, GFix considers moving the o1 to be dangerous and does not fix the bug.

Fourth, GFix decides where to put the defer. If all static o1s are to close c, receive a value from c, or send out the same constant, GFix inserts the defer right after channel c’s declaration (e.g., line 3 in Figure 3). If all o1s send the same variable, GFix checks whether the code site defining the variable dominates all returns. If so, GFix inserts the defer after that site. For all other cases, GFix does not fix the bug.

4.4 Strategy-III: Adding Channel Stop

This strategy is to add channel stop to notify Go-B that Go-A has encountered a problem, so that if Go-B blocks at o2, Go-B can stop its execution. For Go-A, we can defer closing stop in the function that declares c, and the closing operation tells Go-B that Go-A will not conduct any operation on c (including o1) from now on. For Go-B, we change o2 to a select with two cases, the first one is to wait for the original o2, and the second one is to receive a message from stop. We also force Go-B to stop its execution, if it proceeds the second case.

We call bugs that can be fixed by this strategy multiple-operations bugs, since they can be cases where Go-B conducts multiple operations on c (even after o2). When a multiple-operations bug is triggered, Go-A has left the function where c is valid, and no other goroutine interacts with Go-B on c. Thus, all operations on c by Go-B can be skipped.

Figure 4 shows a multiple-operations bug. In each loop iteration, the child goroutine takes a line of inputs at line 7 and sends it to the parent goroutine at line 12 through channel scheduler (the buggy channel). When all inputs are processed, err at line 7 is not nil, so that the child closes channel scheduler at line 9 and stops its execution at line 10. When scheduler is closed, ok at line 22 is false. The parent goroutine returns from function Interactive() and the program successfully continues its execution. However, if the parent receives a message from channel abort at line 21, the bug is triggered and the child blocks at line 12 (o2 in our formalization) forever. Since o2 is in a loop, if we simply unblock one of its executions (e.g., by increasing the buffer size by one), the child will block again at o2 in the next loop iteration.

The patch generated by GFix is also shown in Figure 4. GFix adds channel s stop at line 3, defers closing it at line 4, and replaces o2 at line 12 with the select at lines 13–16. When the parent goroutine returns from function Interactive(), the closing at line 4 is conducted, and the case at line 15 becomes non-blocking. The child proceeds the case and stops its execution.

How to identify multiple-operations bugs? GFix largely reuses the checking mechanisms in the previous two strategies to identify multiple-operations bugs, but there are several differences. First, GFix does not require that Go-B conducts only one operation on c. Second, since a patch in this strategy can unblock multiple goroutines blocking at o2, when identifying Go-B, GFix also considers goroutines created in a loop. Third, GFix leverages return to stop Go-B’s execution (e.g., line 15 in Figure 4), and thus GFix checks whether Go-B conducts o2 in the function used to create Go-B. Fourth, GFix only fixes a bug when instructions after o2 do not generate any side effect beyond Go-B, which is similar to the previous two strategies. However, the difference is that here GFix does not consider operations on c after o2 as having side effects.

5 IMPLEMENTATION AND EVALUATION

5.1 Methodology

Implementation, Configuration, and Platform. We implement both GCatch and GFix using Go-1.14.2. All our static analysis is based on the SSA package [21], which provides sufficient type, data flow, and control flow analysis support. The code transformations in GFix are achieved using the AST package [17], since modified ASTs can be easily dumped to Go source code.

We leverage an existing alias analysis package [20] and modify it to also inspect functions unreachable from main(), since we want to improve code coverage when analyzing Go libraries. The call-graph analysis [18] we use has a limitation, in that when an interface function or a function pointer is called, the analysis reports all functions matching the signature as callees, leading to many false positives. Thus, when the call-graph analysis reports more than one callee for a call site of an interface function or a function pointer, we ignore the results. This design decision may cause GCatch to miss some bugs and report new false positives.

Different strategies in GFix result in patches with different levels of readability or complexity (in terms of lines of changed code). Strategy-I changes only one line of code and generates the simplest patches. Strategy-III adds an extra channel. As a result, Strategy-III patches are most complex and are most difficult to validate. We configure GFix’s dispatcher component (Figure 2) to attempt Strategy-I first, then Strategy-II, and finally Strategy-III, so that GFix generates the simplest possible patch for each input bug. We think this configuration matches how GFix will be used in reality.
Our selected projects cover different types of Go applications (e.g., container systems, web services). They represent typical usage of Go when implementing systems software. Several of the projects are widely used infrastructure systems (e.g., Docker [12], Kubernetes [22]) in cloud environments. It is essential to detect bugs and ensure the correctness for these projects, since their correctness affects all the applications running on top of them. The selected projects are of middle to large sizes, with lines of source code ranging from one thousand to more than three million.

### Evaluation Metrics

We evaluate GCatch and GFix separately. In the case of GCatch, we want to assess its effectiveness, accuracy, and coverage. To do this, we apply GCatch to all packages in the latest versions of the selected applications. We count how many bugs are detected and how many false positives are reported for the effectiveness and the accuracy. However, since we don’t know how many concurrency bugs are in the latest application versions, we cannot use them to measure the coverage. Instead, we conduct a manual study on a released set of Go concurrency bugs [87] and count how many bugs there can be detected by GCatch.

We want to know the correctness, performance, and readability of GFix’s patches. To assess correctness, we manually inspect each patch to determine whether it can fix the bug and whether it changes the original program semantics. To assess performance, for every detected bug, we use all the unit tests that can execute the buggy code to measure the runtime overhead incurred by the patch (i.e., performance impact). We run each unit test ten times with both the patched version and the original version. We compute the overhead using the average execution time. To assess readability, we count how many lines of source code are changed by each patch.

### 5.2 GCatch Results

As shown in Table 1, GCatch finds 149 previously unknown BMOC bugs and 119 previously unknown traditional concurrency bugs. We have reported all of the bugs to developers. At the time of writing, developers have already fixed 125 BMOC bugs and 85 traditional bugs in more recent application versions based on our reporting and confirmed another 22 bugs as real bugs.

<table>
<thead>
<tr>
<th>App Name</th>
<th>BMOC(_C)</th>
<th>BMOC(_M)</th>
<th>Forget Unlock</th>
<th>Double Lock</th>
<th>Conflict Lock</th>
<th>Struct Field</th>
<th>Fatal</th>
<th>Total</th>
<th>S.-I</th>
<th>S.-II</th>
<th>S.-III</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>212</td>
<td>11</td>
<td>83</td>
<td>02</td>
<td>10</td>
<td>25</td>
<td>30</td>
<td>3613</td>
<td>12</td>
<td>-</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>Kubernetes</td>
<td>141</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>-</td>
<td>55</td>
<td>100</td>
<td>3211</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>Docker</td>
<td>497</td>
<td>-</td>
<td>11</td>
<td>23</td>
<td>10</td>
<td>31</td>
<td>-</td>
<td>5613</td>
<td>40</td>
<td>1</td>
<td>6</td>
<td>47</td>
</tr>
<tr>
<td>HUGO</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>01</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>42</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Gin</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>frp</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Gogs</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Syncthing</td>
<td>01</td>
<td>-</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>-</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>etcd</td>
<td>398</td>
<td>-</td>
<td>61</td>
<td>12</td>
<td>01</td>
<td>72</td>
<td>40</td>
<td>5714</td>
<td>24</td>
<td>1</td>
<td>9</td>
<td>34</td>
</tr>
<tr>
<td>v2ray-core</td>
<td>-</td>
<td>01</td>
<td>21</td>
<td>22</td>
<td>30</td>
<td>-</td>
<td>73</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prometheus</td>
<td>21</td>
<td>-</td>
<td>11</td>
<td>11</td>
<td>02</td>
<td>02</td>
<td>-</td>
<td>47</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>fzf</td>
<td>-</td>
<td>-</td>
<td>01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>01</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>traefik</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Caddy</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Go-Ethereum</td>
<td>919</td>
<td>03</td>
<td>41</td>
<td>92</td>
<td>-</td>
<td>67</td>
<td>30</td>
<td>3131</td>
<td>6</td>
<td>-</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Beeego</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>mkcert</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TiDB</td>
<td>10</td>
<td>-</td>
<td>03</td>
<td>30</td>
<td>20</td>
<td>02</td>
<td>-</td>
<td>64</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>CockroachDB</td>
<td>42</td>
<td>-</td>
<td>50</td>
<td>01</td>
<td>21</td>
<td>01</td>
<td>-</td>
<td>1110</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>gRPC</td>
<td>60</td>
<td>-</td>
<td>-</td>
<td>01</td>
<td>10</td>
<td>10</td>
<td>20</td>
<td>101</td>
<td>4</td>
<td>-</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>bbolt</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>40</td>
<td>60</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>147403</td>
<td>25</td>
<td>3213</td>
<td>1916</td>
<td>93</td>
<td>3313</td>
<td>260</td>
<td>268188</td>
<td>99</td>
<td>4</td>
<td>21</td>
<td>124</td>
</tr>
</tbody>
</table>

Table 1: Evaluation Results. Applications are ranked by the number of stars on GitHub. In the GCatch columns, BMOC\(_C\) denotes BMOC bugs only involving channels, BMOC\(_M\) denotes BMOC bugs involving both channels and mutexes, and ‘x’ denotes both bug and false positive numbers are zero. In the GFix columns, S. is short for strategy, and '-' denotes bug number is zero.
allocated to the child goroutine cannot be released. If the child goroutine is repeatedly created and blocked, more system resources will be occupied, which will in turn influence other goroutines and programs running on the same machine.

The largest application in our evaluation (Kubernetes) contains more than three million lines of code. The BMOC detector spends 25.6 hours inspecting all its packages and finds 15 bugs. This analysis time is the longest among all applications we evaluate. These results demonstrate the BMOC detector can scale to large, real Go systems software. For ten small applications we evaluate (e.g., frp, HUGO), the BMOC detector can finish its analysis in less than one minute.

**False positives.** The BMOC detector is accurate. It reports 51 false positives, and the true-bug-vs-false-positive rate is around 3:1. The false positives come from three sources.

Twenty of the false positives are caused by infeasible paths. GCatch only inspects branch conditions involving read-only variables and constants. Nine false positives are caused by path conditions that involve non-read-only variables, and the conjunctions of those conditions are unsatisfiable. For loops whose iteration numbers cannot be computed statically, GCatch unrolls them at most twice. If this is wrong, sending/receiving operations inside those loops will be counted incorrectly. The remaining 11 false positives are due to this reason.

Another 17 false positives result from limitations of the alias analysis. When a channel is sent to a goroutine through another channel, the alias analysis cannot determine whether the received channel is the same as the sent channel. Thus, GCatch fails to figure out how a channel operation at the sender goroutine is unblocked by the receiver. This is the reason for 15 false positives. The remaining two alias-analysis false positives are cases in which a channel is saved into an array, the blocking operation is conducted by referring to the array element, but the unblocking operation refers to the channel variable. The alias analysis fails to identify the two operations as being conducted on the same channel. Thus, GCatch fails to figure out how the blocking operation is unblocked for the two cases.

Finally, 14 false positives are due to limitations of the call-graph analysis. Sometimes, the call-graph analysis fails to identify caller functions for a callee, such as when the callee implements an interface function. If the callee contains blocking operations, GCatch may not know how the blocking operations can be unblocked, leading to false positives.

It is not difficult to differentiate real bugs from false positives. For each detection result, GCatch provides information about the buggy channel, the blocking operations, the path combination, related call chains, and the analysis scope. We take two steps to inspect the information. First, we check whether the channel is used only as the primitive to conduct a channel operation (e.g., sending, receiving). If so, the results of alias analysis are correct. For almost all reported results, the channel is used only in this way and it is used in less than ten places. There are very few cases where a channel variable is used for other purposes (e.g., being sent through another channel). We will further check whether the alias results are correct for those cases and how the alias results impact GCatch. Second, we check whether execution paths in the path combination and the call chains are feasible. Since they are within the channel’s analysis scope, they are usually very short. Based on our own experience, it takes a GCatch user roughly five minutes to analyze a reported result and decide whether it is a real bug or a false positive.

**Coverage.** There are 49 BMOC bugs in the public Go concurrency bug set [67]. We conduct a manual study on these bugs and find that GCatch can detect 33 of them. Since the bug set is a random sample of real-world Go concurrency bugs, the study result shows that GCatch has a good coverage (i.e., 67%) of real-world BMOC bugs. GCatch fails to detect the other bugs due to the following reasons.

First, GCatch misses two BMOC bugs caused by conducting a channel operation in a critical section. The reason is that the identified LCA function (Section 3.2) is called by the function containing the critical section. GCatch only inspects operations in the LCA function and its callees, and thus it misses the locking operation protecting the critical section.

Second, some bugs can only be detected with dynamic information. For example, there are three etcd bugs where a goroutine waits to receive a particular value from a channel. If the received value is not the right one, the goroutine waits to receive from the channel again. However, GCatch does not know the waited-for value cannot be sent out statically.

Third, GCatch does not model some concurrency primitives (e.g., WaitGroup, Cond) and libraries (e.g., time), and thus it fails to detect bugs caused by those primitives and libraries.

Fourth, GCatch does not conduct any data flow analysis, so that it misses two bugs caused by assigning nil to a channel and then sending a value to the channel, since sending to a nil channel blocks a goroutine forever.

**Evaluating the disentangling policy.** We apply path enumeration (Section 3.3) and BMOC bug detection (Section 3.4) to function main() of each evaluated application directly without applying the disentangling policy (Section 3.2). Compared with when GCatch analyzes the package containing function main(), on average, disabling disentangling leads to over 115X slowdown and 0.59X memory-usage increase.

After inspecting the execution log, we find that disentangling improves the performance of GCatch in two ways: 1) it significantly shortens enumerated execution paths, since those paths are inside channels’ analysis scopes, and 2) it significantly reduces the number of constraints in constraint conjunctions analyzed by Z3, because each channel is analyzed only with a few other primitives.

**Results of the other five detectors.** As shown in Table 1, the five traditional detectors find 119 previously unknown bugs.

The traditional detectors report 67 false positives mainly for four reasons. 18 false positives are due to semantic reasons. For example, some functions are actually a wrapper of a locking operation, and the acquired lock is released after the end of the function, but GCatch mistakenly reports them as a lock-without-unlock. There are 17 false positives caused by infeasible paths. Another nine false positives are due to errors when computing alias information for mutex variables. Finally, nine false positives are caused by failing
to consider the calling context when computing the lockset for a
struct field access.

5.3 GFix Results

Overall results. GCatch reports 147 BMOC bugs involving channels
only (column “BMOC C” in Table 1), which are targets of GFix. In
total, GFix generates patches for 124 of them. How the fixed
bugs distribute across different strategies is shown in Table 1. We
make our best effort in evaluating patches’ correctness through
code inspection and injecting random-length sleeps around the
channel operations causing each bug. We confirm that all generated
patches are correct, and that they can fix the bugs without changing
the original program semantics. We have provided the generated
patches when reporting the bugs. So far, developers have applied
87 of these patches directly.

GFix decides not to fix 23 of the bugs for four reasons. For nine
of the bugs, the blocking coroutine is the parent, and thus GCatch
decides not to fix them. For another ten bugs, there are side effects
after the o2 instruction, and if GFix unblocks the o2, the program
semantics may be changed. In one bug, the o1 instruction is a
receiving operation and the received value is used, so that GFix
cannot apply Strategy-II to defer the receiving. The remaining three
bugs involve one or more than two goroutines.

Patches’ performance. For 116 out of the 124 bugs fixed by GFix,
we find unit tests that can execute their buggy code. Thus, we
measure the runtime overhead after applying the patches for them.
The runtime overhead is small. The average overhead is 0.26%, the
maximum is 3.77%, and there are only 14 bugs with overhead larger
than 1%. These results show that GFix’s patches incur a negligible
performance impact.

Patches’ readability. We measure the readability of GFix’s patches
by counting the changed lines of source code. Changing lines of
source code includes adding new lines, removing existing lines, and
replacing existing lines with new lines. For example, the patch in
Figure 1 changes one line, and the patch in Figure 3 changes four
lines (three added and one removed).

On average, GFix changes 2.67 lines of code to fix a bug. There
are 99 bugs fixed by Strategy-I. For all of them, GFix changes only
one line of code. For the four bugs fixed by Strategy-II, GFix changes
four lines of code each. Strategy-III is the most complex strategy.
On average, GFix changes 10.3 lines for each bug fixed by this
strategy, and the largest patch involves 16 changed lines. Overall,
GFix changes very few lines of code to fix a bug, and its patches
have good readability.

GFix’s execution time. On average, GFix takes 90 seconds to gen-
erate a patch. The patch generation time varies across different
applications. Kubernetes is the largest application in our evaluation.
On average, GFix spends 178 seconds generating a patch for it,
which is the longest among all applications. Docker also contains
more than one million lines of code. The average patch genera-
tion time for Docker is 151 seconds, and it is the second longest.
Bolt only contains ten thousand lines of code, and GFix takes the
shortest average time to generate a patch for it (2.8 seconds).

After examining GFix’s execution time, we find that GFix spends
most of that time (i.e., 98%) converting an input program into SSA,
constructing the program’s call graph, and computing the alias rela-
tionships, since GFix relies on this information for its functionality.
After those preprocessing steps, GFix can figure out which fixing
strategy to use and conduct the corresponding code transformation
very quickly (1.9 seconds on average).

6 DISCUSSION AND FUTURE WORK

GCatch detects blocking misuse-of-channel (BMOC) bugs in Go,
while GFix synthesizes patches for the detected BMOC bugs us-
ning channels and channel-related language features (e.g., select,
defer). The two techniques construct an end-to-end system for
eliminating BMOC bugs in real Go systems software. In this section,
we briefly discuss the limitations of GCatch and GFix, future work
to improve them, and how to extend them to other programming
languages.

Limitations and Future Work of GCatch. The constraint system
in GCatch models only channels and mutexes, and thus GCatch
can only detect BMOC bugs caused by them. However, after chang-
ing other primitives to channels (as we did for mutexes), GCatch
can detect blocking bugs caused by these primitives as well. For
example, to model a condition variable (Cond), we can change it
to an unbuffered channel, its Wait() operations to receiving from
the channel, its Signal() operations to sending to the channel in
a select with a default clause, and its Broadcast() operations
repeatedly sending to the channel using a loop that contains a
select with a sending operation and a default clause.

Misuse of channels can also cause non-blocking bugs. Another di-
rection to extend GCatch is to model and detect when non-blocking
misuse-of-channel bugs happen. For example, sending to an already
closed channel triggers a panic. We can enhance GCatch to detect
bugs caused by this error by configuring a new type of bug con-
straints where a sending operation has a larger order variable value
than a closing operation conducted on the same channel.

GCatch does report some false positives. Many of these are due
to the limitations of the static analysis packages we use (e.g., alias
analysis, call-graph analysis). Future work can consider improving
classic static analysis for Go, which will in turn benefit GCatch. For
example, alias analysis can be enhanced to identify alias relation-
ships that are generated by passing an object through a channel,
which will help reduce GCatch’s false positives.

Limitations and Future Work of GFix. Although GFix fixes only
those bugs that fit three specific patterns, our experimental results
show that it can still patch the majority of BMOC bugs detected by
GCatch. In the future, we will extend GFix with additional fixing
strategies, such as covering more buggy code patterns and using
other primitive types. Although in our experiments we manually
validate the patches’ correctness, we hope to automate this process.
We leave the design of an automated patch testing framework for
Go to future work.

Generalization to Other Programming Languages. Go covers
many concurrency features (e.g., channel, select) in many other
new programming languages (e.g., Rust, Kotlin), and thus our tech-
niques can potentially be applied to those languages after some
design changes. For example, Rust supports both synchronous channels, which are identical to Go’s buffered channels, and asynchronous channels, which have unlimited buffers. To enable GCatch on Rust, we don’t need to change GCatch’s constraint system for Rust’s synchronous channels, but we do need to extend it to allow sending operations to always proceed for Rust’s asynchronous channels. As another example, Kotlin organizes threads in a hierarchical way, and when a parent thread stops due to cancellation or exception, all its children are terminated. Thus, to apply GCatch to Kotlin, we need to extend GCatch’s proceeding and blocking constraints by considering how parent threads execute for their child threads.

7 RELATED WORK

**Empirical Studies on Go.** Researchers have performed several empirical studies on Go. Ray et al. [77] inspect the correlation between different types of bugs and different programming languages (including Go). Dilley et al. [34] examine how channels are used by analyzing 865 Go projects. Tu et al. [87] systematically study concurrency bugs in Go. As we discussed in Section 2.2, their findings have inspired the design of GCatch and GFix.

**Static Concurrency Bug Detection.** Many research works have been conducted to statically detect deadlocks [49, 58, 75, 82] and data races [54, 55, 67, 68, 74, 76, 89] in C/C++ and Java. Although these algorithms promise to detect similar concurrency bugs in Go, none of them are designed for concurrency bugs related to channels. Fahndrich et al. [36] design a technique to identify deadlocks among processes communicating through channels. However, a channel in their context has no buffer, is only shared by two processes, and has fewer channel operations than a Go channel. Thus, their technique is not enough to detect BMOC bugs for Go.

Several blocking bug detection techniques are designed for Go [39, 59, 60, 70, 80]. These techniques extract a Go program’s execution model by inspecting channel operations (and locking operations); they then apply model checking to prove the liveness of the execution model or identify bugs. These techniques need to inspect the whole input program from function main() and consider primitives altogether. Thus, they have severe scalability issues. However, GCatch models channel operations using a novel constraint system. In addition, it disentangles primitives of an input program to scale to large Go programs, even those with millions of lines of code.

There are two static bug detection tool suites built for Go: staticcheck [23], and the built-in vet tool [9]. Each of them contains four different concurrency-bug detectors. Unfortunately, those detectors cover very specific buggy code patterns (e.g., deferring a locking operation right after a locking operation on the same mutex), and none of them aim to detect bugs caused by errors when using channels. Thus, we don’t apply the two suites to the benchmark programs used in our evaluation and compare GCatch with them directly. Instead, we manually examine each bug identified by GCatch and count how many of them can be detected by the two tool suites. Overall, the two suites can detect 0 out of 149 BMOC bugs and 20 out of 119 traditional concurrency bugs reported by GCatch. The 20 bugs that the two tool suites detect are all caused by calling testing.Fatal() in a child goroutine.

**Dynamic Concurrency Bug Detection.** Many dynamic detection techniques are designed for traditional programming languages (e.g., C/C++, Java), and they can effectively identify concurrency bugs caused by errors when accessing shared memory, like data races [35, 43–46, 56, 62, 79, 81, 94, 95], atomicity violations [28, 37, 65, 71, 84, 90], order violations [40, 66, 93, 96] and deadlocks [26, 27]. Unfortunately, those techniques are not designed to identify channel-related concurrency bugs.

Go provides two built-in dynamic detectors for deadlocks and data races, respectively [11, 19]. However, a recent empirical study shows that these two detectors can only identify a small portion of blocking and non-blocking bugs in real Go programs [87].

There are dynamic detectors aiming to identify deadlocks in MPI programs [38, 41, 42, 83, 88]. Although MPI programs also rely on message passing for thread communication, their message-passing model (actor model [7]) is different from Go’s message-passing model (CSP model [10]). For example, a sending (or receiving) operation in MPI needs to specify the process ID of the receiver (or sender), while a sending (or receiving) operation in Go is conducted through a reference to a channel. In addition, the techniques built for MPI programs fail to model many important channel operations in Go (e.g., close(), select). Thus, these detectors cannot effectively detect channel-related blocking bugs in Go.

Moreover, the effectiveness of dynamic techniques largely depends on inputs to run the program and the observed interleavings. In contrast, static techniques do not rely on inputs and have better code and interleaving coverage. Therefore, we choose to build GCatch based on static analysis.

**Concurrency Bug Fixing.** Existing techniques rely on controlling thread scheduling to fix or avoid deadlocks caused by locking operations [53, 91, 92, 97]. There are also techniques that fix non-blocking bugs, and they achieve their goal by disabling bad timing of accessing a shared resource [51, 52, 61, 64] or by eliminating the sharing altogether [47]. Unlike these techniques, GFix (Section 4) focuses on channel-related blocking bugs and generates patches with good readability by leveraging Go’s unique language features (e.g., defer, select).

8 CONCLUSION

This paper presents a new BMOC bug detection technique GCatch and a new BMOC bug fixing technique GFix for Go. GCatch provides an effective disentangling strategy to separate concurrency primitives of each input Go software and a novel constraint system to model channel operations in Go. GFix contains three fixing strategies to patch bugs detected by GCatch using Go’s unique language features. In the experiments, GCatch finds more than one hundred previously unknown BMOC bugs in large infrastructure software and GFix successfully patches most of them. Future research can further explore how to detect and fix other types of concurrency bugs for Go.

ACKNOWLEDGEMENT

We would like to thank Lin Tan, our shepherd, and the anonymous reviewers for their insightful feedback and comments. We are also grateful to Victor Fang from AnChain.AI and Heqing Huang from TensorSecurity for valuable discussion on potential usage of our techniques. This work is supported in part by NSF grant CNS-1955965.