# Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems

## Abstract

*This paper describes GCatch, a static Go concurrency bug detection system, and GFix, an automated Go concurrency bug fixing system.*

## 1. Introduction

Go is a new programming language designed for efficient and reliable concurrent programming [7]. For this purpose, Go provides lightweight threads (called goroutines) and advocates the use of channel for thread communication. In recent years, Go has become increasingly popular [15, 17] and has already been widely adopted to build systems software in production environments (*e.g.*, Docker [6], Kubernetes [13], gRPC [3]).

Unfortunately, there are still many concurrency bugs in Go [67, 75]; these bugs are the most difficult to debug [53, 56], and they severely hurt the reliability of multi-threaded software systems [48, 63].

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 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 `Exec()`. No other goroutine can pull messages from `outDone`, leaving the child 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. With the increasing adoption of Go, it is urgent to fight concurrency bugs in Go, especially those caused by misuse of channel, since Go advocates the use of channel and many developers choose Go because of its good support for channel [45, 61].

However, channel-related concurrency bugs cannot be effectively detected by existing techniques, due to their different design goals [25, 26, 35, 44, 57, 58, 65, 71, 72], limited bug coverage [4, 5, 10], and scalability issues [27, 46, 47, 59, 69]. Advanced techniques designed specifically for channel-related bugs in Go are needed.

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

**Figure 1: A previously unknown Docker bug and its patch.**

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, since the majority of channel-related bugs in Go are blocking bugs [75]. Its design consists of two components: a disentangling policy that divides concurrency primitives into small groups and enables bug detection to focus on each small group in a small program scope, and a novel constraint system that models channel-related concurrency features in Go and transforms BMOC bug detection to constraint solving. Channel has states (*e.g.*, full or not), making it more difficult to model than other primitives (*e.g.*, mutex). Besides the BMOC detector, GCatch also contains five additional concurrency bug detectors.

A BMOC bug will continue to hurt the system's reliability until it is fixed. Thus, we further design an automated fixing system, GFix, that leverages Go's channel-related concurrency features (*e.g.*, channel, `select`) to patch BMOC bugs detected by GCatch (Figure 2). GFix conducts static analysis to categorize input BMOC bugs into three groups and provides different strategies for each. Unlike existing techniques that prevent lock-related deadlocks at runtime [39, 78, 79, 84], GFix aims to fix BMOC bugs using patches with good readability.

The bug and its patch 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 chooses the second `case`, the child blocks at line 7 endlessly, which is exactly the root cause. 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 153 previously unknown BMOC bugs and

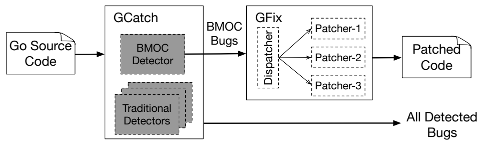

**Figure 2: The workflow of GCatch and GFix.**

123 traditional bugs. GFix generates patches for 129 detected BMOC bugs. All patches are correct, most of them incur less than 1% runtime overhead, and on average, each patch changes 2.7 lines of code. 85 of the generated patches have been applied by developers directly.

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. This 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.

## 2. Background

This section gives some background of this project, including Go's synchronization primitives, the 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 [75]. It can send data across goroutines and synchronize different goroutines to implement complex functionalities [16, 62]. 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 buffer still has empty slots. If a channel is closed, a goroutine seeking to receive data 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. The 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 non-deterministically chooses one.

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

### 2.2. Concurrency Bugs in Go

Unfortunately, concurrency bugs are more likely to happen in Go [67]. Thus, we need to detect them and fix them. A previous empirical study [75] 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 channel are blocking bugs [75], we concentrate our efforts on blocking misuse-of-channel (BMOC) bugs. According to the study, BMOC bugs are usually caused by errors when using channel alone or using channel together with mutex.

### 2.3. Existing Constraint Systems

Previously, constraint solving was used to dynamically detect concurrency bugs [31, 43, 77]. 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 when a bug (*e.g.*, data race, atomicity violation) happens (bug constraints).

Unfortunately, existing constraints do not cover how channel operations proceed, and thus they cannot detect channel-related bugs. Channels have states (*i.e.*, the number of elements in them, closed or not), so that they are more complex to model than other primitives without states (*e.g.*, mutexes). There are two challenges in modeling channel operations using constraints. 1) How to model a channel's states? 2) How to model state updates for a program execution? We will discuss how we solve the two challenges 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 and a novel constraint system to 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 other

**Algorithm 1** BMOC Bug Detection

```
Require: An input Go program ($P$)
1: function ($P$)
2:     $C_{\text{graph}} \leftarrow$ BuildCallGraph($P$)
3:     $Alias \leftarrow$ ConductAliasAnalysis($C_{\text{graph}}$)
4:     $Primitives \leftarrow$ SearchSynPrimitives($C_{\text{graph}}$)
5:     $OP_{\text{map}} \leftarrow$ SearchSynOperations($Primitives$, $C_{\text{graph}}$, $Alias$)
6:     $D_{\text{graph}} \leftarrow$ BuildDependGraph($OP_{\text{map}}$, $C_{\text{graph}}$)
7:     for each channel $c$ in $Primitives$ do
8:         $scope$, $P_{\text{set}} \leftarrow$ DisentanglingAnalysis($c$, $OP_{\text{map}}$, $D_{\text{graph}}$)
9:         $GO_{\text{set}} \leftarrow$ SearchGoroutines($c$, $scope$)
10:        $PCs \leftarrow$ ComputePathCombination($GO_{\text{set}}$, $P_{\text{set}}$, $OP_{\text{map}}$, $scope$)
11:        for each path combination $pc$ in $PCs$ do
12:            $Groups \leftarrow$ ComputeSuspiciousOpGroups($pc$, $P_{\text{set}}$, $GO_{\text{set}}$)
13:            for each group ($g$) in $Groups$ do
14:                $\Phi_R \leftarrow$ ReachConstraint($g$, $P_{\text{set}}$)
15:                $\Phi_B \leftarrow$ BlockConstraint($g$)
16:                $\Phi \leftarrow \Phi_R \wedge \Phi_B$
17:                if Z3 finds a solution ($s$) for $\Phi$ then
18:                    ReportBug($g$, $s$)
19:                end if
20:            end for
21:        end for
22:    end for
23: end function
```

detectors for concurrency bugs that also appear in classic programming languages (*i.e.*, "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 7–22 in Algorithm 1). Given a channel $c$, GCatch only inspects $c$ and a few other related primitives ($P_{\text{set}}$) in a small program scope (*scope*). We will discuss the disentangling policy in Section 3.2.

Third, GCatch determines the goroutine set ($GO_{\text{set}}$) that accesses channel $c$ by inspecting all goroutines created in $c$'s analysis scope (line 9 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 created at line 5 as accessing `outDone`.

Fourth, GCatch computes path combinations by enumerating all possible execution paths for each goroutine in $GO_{\text{set}}$ (line 10 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 12. Each suspicious group contains operations that together can potentially block forever.

Sixth, GCatch uses Z3 [22] to examine all suspicious groups (lines 13–20 in Algorithm 1). For each group, GCatch computes the constraints ($\Phi_R$) for the program to execute just before the group operations and the constraints ($\Phi_B$) for all group operations to block. If Z3 finds a solution for the conjunction of $\Phi_R$ and $\Phi_B$ at line 17, 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 7–22 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 (*scope* at line 8). 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 8) must be analyzed together with $c$.

***How to compute 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 *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 functions and consider *scope* of c as the union of the computed scopes.

***How to compute $P_{set}$?*** GCatch builds a graph ($D_{\text{graph}}$) at line 6 in Algorithm 1 by computing the dependence relationship for each pair of primitives. Later, at line 8, 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` can*not* 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 $c$.

The $P_{\text{set}}$ of $c$ contains $c$ and all other 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 `ctx.Done()` and `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`. They two will be inspected together when GCatch examines `ctx.Done()`.

## 3.3. Computing Path Combinations

For each goroutine in $GO_{\text{set}}$, GCatch conducts an inter-procedural, depth-first search to enumerate all its execution paths within *scope* (line 10 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.

GCatch enumerates all possible path combinations for all goroutines in $GO_{\text{set}}$. GCatch filters out combinations that are either infeasible or unlikely to cause any BMOC bug. For example, GCatch inspects branch conditions only involving real-only variables and constants and filters out path combinations with conflicting 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 terminating condition but different loop iterations, 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` with two cases at line 9 and an `if` at line 11, GCatch finds three possible paths for the parent. 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 12 in Algorithm 1) with 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 ($\Phi_{\text{B}}$) (lines 13–20).

The current version of GCatch only models channel operations and mutex operations, since these cause most BMOC bugs [75]. 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 channel elements before the operation. For example, $CB_{si}$ indicates that before the sending at line i, $CB_{si}$ elements are in the channel. The value of a *CB* variable is computed as the number of sendings minus the number of receivings, 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.

***How to compute $\Phi_R$?*** $\Phi_{\text{R}}$ represents all the necessary constraints for goroutines in $GO_{\text{set}}$ to execute just before operations in a suspicious group. Besides variables for channel states, $\Phi_{\text{R}}$ contains another two types of variables. First, we associate each instruction with an *O* variable denoting its execution order, *e.g.*, $O_i$ representing the order of the instruction at line i. 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. For instance, $P_{(\text{si, rj})} = 1$ means the sending at line i unblocks the receiving at line j. When $P_{(\text{si, rj})} = 1$, $s_{\text{i}}$ and $r_{\text{j}}$ execute at the same time (*i.e.*, $O_{\text{si}} = O_{\text{rj}}$).

$\Phi_{\text{R}}$ 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. GCatch only considers operations belonging to primitives within $P_{\text{set}}$ for $\Phi_{\text{sync}}$. Since $\Phi_{\text{order}}$ and $\Phi_{\text{spawn}}$ were discussed in previous literatures [34, 77], we only explain how to compute $\Phi_{\text{sync}}$.

Channel. GCatch models sending operations, receiving operations, and `select` statements as follows.

If a sending operation at line i ($s_{\text{i}}$) 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 receivings executed by different goroutines on the same channel. The computed constraints for $s_{\text{i}}$ are written as follows:

$$CB_{si} < BS \vee [(\bigvee_{\forall rx \in R} P_{(si,rx)} = 1 \wedge O_{si} = O_{rx}) \wedge \sum_{\forall rx \in R} P_{(si,rx)} = 1]$$

Similarly, the proceeding of a receiving operation at line j ($r_{\text{j}}$) 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 sendings executed by different goroutines on the same channel. The constraints for $r_{\text{j}}$ are written as follows:

$$CB_{rj} > 0 \vee CLOSED_{rj} \vee [(\bigvee_{\forall sx \in S} P_{(sx,rj)} = 1 \wedge O_{sx} = O_{rj}) \wedge \sum_{\forall sx \in S} P_{(sx,rj)} = 1]$$

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.
Mutex. 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) $\Phi_{order}$ requires that instructions before the group operations must have smaller $O$ variable values, compared with the group operations, and 2) $\Phi_{sync}$ 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 is 3-5-9-13-14-vs-6-7. Since `ctx.Done()` is not in `outDone`'s $P_{set}$, only the sending at line 7 satisfies the requirements for being in a suspicious group. Thus, GCatch only finds one suspicious group containing line 7.

For $\Phi_R$, $\Phi_{order}$ is $(O_3 < ... < O_{13} < O_{14} \wedge O_6 < O_7)$ which is enforced by the two paths in the combination, $\Phi_{spawn}$ is $(O_5 < O_6)$ required by the goroutine creation at line 5, and $\Phi_{sync}$ is empty, since there is no operation belonging to any primitive in the $P_{set}$ on the path of the parent and on the path of the child before line 7. For $\Phi_B$, $\Phi_{order}$ is $(O_3 < O_7 \wedge ... \wedge O_{14} < O_7 \wedge O_6 < O_7)$, and $\Phi_{sync}$ is $(CB_{s7} = BS)$. Since `outDone` is an unbuffered channel, $BS$ is 0. Z3 finds a solution that is $(O_3 = 0 \wedge ... \wedge O_{14} = 4 \wedge O_6 = 5 \wedge O_7 = 6 \wedge CB_{s7} = 0)$, which means when the program executes in the order $3 \rightarrow ... \rightarrow 14 \rightarrow 6 \rightarrow 7$, the child blocks at line 7 forever.

### 3.5. Traditional Checkers

GCatch contains five additional checkers to detect three types of Go concurrency bugs. We briefly discuss them as follows.

GCatch detects traditional deadlocks caused by misuse of mutex by conducting intra-procedural, path-sensitive analysis to identify lock-without-unlocks, and inter-procedural, path-sensitive analysis to identify double locks and deadlocks that result from acquiring two locks in conflicting orders.

Similar to previous techniques designed for C [64, 65], GCatch implements an intra-procedural, path-sensitive algorithm to collect lockset information for each struct field access. If a field is protected by a lock for most accesses, GCatch reports the accesses without such protection as data races.

Detecting violations of API usage rules is effective at discovering bugs in classic programming languages [21, 36, 51]. GCatch detects concurrency bugs caused by errors when using a suite of APIs in the `testing` package (*e.g.*, `Fatal()`).

## 4. Fixing Go Concurrency Bugs

This section describes GFix in detail. As shown in Figure 2, GFix takes BMOC bugs detected by GCatch as input. Its dispatcher component leverages static analysis to categorize input bugs, and the corresponding patching component conducts source-to-source transformation to fix the bugs.

The design philosophy of GFix is to leverage Go's channel-related concurrency features to fix bugs. Those features are powerful and frequently used by Go programmers. As a result, GFix's patches slightly change the buggy programs (in terms of lines of code) and follow what developers usually do in reality, and thus the patches have good *readability* and are easy to be validated and accepted by developers.

### 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 *o1* on *c*, causing *Go-B* to be blocked at another operation *o2* on *c* forever. After fixing the bug, *Go-B* can continue its execution after *o2* or just stop its execution at *o2*.

To guarantee a patch's correctness, GFix must make sure there is no race condition between instructions before *o1* in *Go-A* and instructions after *o2* in *Go-B*, if *Go-B* continues its execution, since the patch removes the order enforced by *o1* and *o2*. 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*.[1]

GFix provides three fixing strategies based on three different angles 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. Next, we will discuss how GFix identifies bugs for each strategy and the corresponding code transformation.

[1] We will use the parent (child) goroutine and *Go-A* (*Go-B*) exchangeably in this section.

## 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 [74]. 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 executing function `Exec()`, *Go-B* is the child created at line 5, and channel `outDone` is the buggy channel. *Go-B* conducts only one sending operation on `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 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*.

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*.

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

```
1  func TestRWDialer(t *testing.T) {
2    stop := make(chan struct{})
3  + defer func() {
4  +   stop <- struct{}{}
5  + }()
6    go Start(stop)
7    conn, err := d.Dial(...)
8    if err != nil {
9      t.Fatalf("dial␣error")
10   }
11 - stop <- struct{}{}
12 }

14 func Start(stop struct{}{}) {
15   ...
16   <-stop
17 }
```

**Figure 3: A missing-interaction bug in etcd.**

not fix the bug. For example, in Figure 1, the child 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 *o1*s if they exist. Figure 3 shows an example. When `t.Fatalf()` 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 static *o1*s, 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 synchro-

```
 1 func Interactive() {
 2   scheduler = make(chan string)
 3+  stop = make(chan struct{})
 4+  defer close(stop)
     ...
19   for {
20     select {
21     case <-abort: return
22     case _, ok := <-scheduler:
23       if !ok { return }
24     }
25   }
26 }
```

```
 5 go func() {
 6   for {
 7     line, err := Input()
 8       if err != nil {
 9         close(scheduler)
10         return
11       }
12-    scheduler <- line
13+    select {
14+    case scheduler <- line:
15+    case <- stop: return
16+    }
17   }
18 }()
```

**Figure 4: A multiple-operations bug in Go-Ethereum.**

nization 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 *o1*s are to close *c*, receive a value from *c*, or to send out the same constant, GFix inserts the `defer` right after channel *c*'s declaration (*e.g.*, line 3 in Figure 3). If all *o1*s 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 to close *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 is to wait for the original *o2*, and the second 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 sends a line of inputs to the parent 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 `scheduler` at line 9 and stops its execution at line 10. When `scheduler` is closed, `ok` at line 22 is `false`. The parent returns from `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 `stop` at line 3, defers to close it at line 4, and replaces *o2* at line 12 with the `select` at lines 13–16. When the parent returns from `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 and Platform.*** We implement both GCatch and GFix using Go-1.14.2. All our static analysis is based on the SSA package [12], which provides sufficient type, data flow, and control flow analysis support. The code transformations in GFix are achieved using the AST package [8], since modified ASTs can be easily dumped to Go source code.

We mainly leverage an existing alias analysis package [11] and modify it to also inspect functions unreachable from `main()`, since we want to improve the code coverage when analyzing Go libraries. The call-graph analysis [9] we use has a limitation, in that when an interface function 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, we ignore the results.

All our experiments are conducted on several identical machines, with Intel(R) Core(TM) i7-7700 CPU, 32GB RAM and Ubuntu-16.04.

***Benchmarks.*** We selected Go projects on GitHub for our evaluation. We first inspected the top 20 most popular Go projects on GitHub (based on the number of stars). Of these, one project is a list of Go projects [1] and another is an ebook on Go [2]. Neither contains any code. Thus, we excluded these two from our evaluation. We also chose all of the six Go projects used in an empirical study on Go concurrency bugs [75]. Three of the six projects were already included in the GitHub top 20 list. Thus, we selected a total of 21 Go projects for our evaluation.

Table 2 in the Appendix shows the details of our selected projects. 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

| App Name | GCatch BMOC$_C$ | BMOC$_M$ | Forget Unlock | Double Lock | Conflict Lock | Struct Field | Fatal | Total | GFix S.-I | S.-II | S.-III | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Go | $21_1$ | $1_1$ | $9_3$ | $0_2$ | $1_0$ | $2_5$ | $3_0$ | $37_{12}$ | 12 | - | 2 | 14 |
| Kubernetes | $16_5$ | $1_0$ | $1_0$ | $1_0$ | - | $7_4$ | $10_0$ | $36_9$ | 11 | - | - | 11 |
| Docker | $49_{10}$ | - | $1_1$ | $2_3$ | $1_0$ | $3_1$ | - | $56_{15}$ | 40 | 1 | 6 | 47 |
| HUGO | - | - | $2_0$ | $0_1$ | - | $2_1$ | - | $4_2$ | - | - | - | - |
| frp | - | - | $1_0$ | - | - | - | - | $1_0$ | - | - | - | - |
| Syncthing | $0_1$ | - | $3_1$ | - | - | $1_2$ | - | $4_4$ | - | - | - | - |
| etcd | $39_6$ | - | $6_1$ | $1_2$ | $0_1$ | $7_2$ | $4_0$ | $57_{12}$ | 24 | 1 | 9 | 34 |
| v2ray-core | - | $0_1$ | - | $2_1$ | $2_1$ | $3_0$ | - | $7_3$ | - | - | - | - |
| Prometheus | $3_0$ | - | $1_1$ | $1_1$ | $0_2$ | $0_2$ | - | $5_6$ | 2 | - | 1 | 3 |
| fzf | - | - | $0_1$ | - | - | - | - | $0_1$ | - | - | - | - |
| Go-Ethereum | $10_{19}$ | $0_3$ | $4_1$ | $9_1$ | - | $6_7$ | $3_0$ | $32_{31}$ | 7 | - | 2 | 9 |
| Beego | - | - | - | - | - | $3_0$ | - | $3_0$ | - | - | - | - |
| TiDB | $1_0$ | - | $0_6$ | $3_0$ | $2_0$ | $0_2$ | - | $6_8$ | 1 | - | - | 1 |
| CockroachDB | $4_2$ | - | $5_0$ | $0_4$ | $2_1$ | $0_3$ | - | $11_{10}$ | 1 | 2 | - | 3 |
| gRPC | $6_0$ | - | - | $0_1$ | $2_0$ | $1_0$ | $2_0$ | $11_1$ | 4 | - | 1 | 5 |
| bbolt | $2_0$ | - | - | - | - | - | $4_0$ | $6_0$ | 1 | - | 1 | 2 |

Table 1: **Evaluation Results.** ***In the GCatch columns, BMOC$_C$ denotes BMOC bugs only involving channel, BMOC$_M$ denotes BMOC bugs involving both channel and mutex, $x_y$ denotes $x$ real bugs and $y$ false positives, and '-' denotes both bug and false positive numbers are zero. In the GFix columns, S. is short for strategy, and '-' denotes bug number is zero. Five applications (Gin, Gogs, traefik, Caddy and mkcert) without detected bugs and false positives are skipped in the table.***

(*e.g.*, Docker [6], Kubernetes [13]) 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 [75] 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 276 previously unknown bugs. We have reported all of the bugs to developers. At the time of the writing, developers have already fixed 199 reported bugs in more recent application versions and confirmed another 24 bugs as real bugs.

***Results of the BMOC detector.*** The BMOC detector finds 153 new bugs from ten different applications. Among them, only two are caused by misuse of channel and mutex (column "BMOC$_M$" in Table 1), while all the others involve channel only (column "BMOC$_C$"). The large number of identified bugs demonstrates the effectiveness of the BMOC detector. For most of the detected BMOC bugs, a child goroutine is blocked forever and system resources allocated to the child cannot be released. If the child goroutine is repeatedly created and blocked, more system resources will be occupied, influencing other goroutines and programs on the same server.

The largest application in our evaluation (Kubernetes) contains three million lines of code. The BMOC detector spends 12 hours to finish inspecting all its packages and finds 17 bugs. The analysis time is the longest among all applications. These results demonstrate the BMOC detector can scale to large, real Go systems software. For small applications (*e.g.*, frp), the BMOC detector can finish its analysis in less than one minute.

False positives. The BMOC detector is accurate. It reports 49 false positives, and the true-bug-vs-false-positive rate is larger than 3:1. The false positives come from three sources.

18 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 cannot be satisfied and involve non-read-only variables. Although GCatch tries its best to identify loops with equivalent conditions, some loop conditions are too complex to figure out statically. The remaining nine false positives are due to path combinations involving two loops with equivalent loop conditions but different loop iteration numbers.

Another 16 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 they can be unblocked, leading to false positives.

Finally, 15 false positives (all in Go-Ethereum) result from a limitation 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.

It is not difficult to differentiate real bugs from false positives. For each reported result, GCatch also provides the path combination and related call chains. Given that each path (and call chain) is within a channel's analysis scope and is not long, it is not time-consuming to identify false positives due to infeasible paths (or incorrect caller-callee relationships).

Coverage. There are 49 BMOC bugs in the public Go concurrency bug set [75]. 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 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 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 the path enumeration (Section 3.3) and the BMOC bug detection (Section 3.4) to function `main()` of each application directly without conducting the disentangling (Section 3.2) . Compared with when GCatch analyzes the package with `main()`, on average, disabling disentangling slows down the analysis 500X and increases the memory usage 1.89X. We analyze execution trace and find that the benefit of disentangling mainly comes from significantly reducing the enumerated paths' lengths and reducing the number of constraints sent to Z3.

***Results of the other five detectors.*** As shown in Table 1, the five traditional detectors find 123 previously unknown bugs.

The traditional detectors report 65 false positives mainly for three reasons. 17 false positives are caused by infeasible paths. Another 16 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. 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 151 BMOC bugs involving channel only (column "BMOC$_C$" in Table 1), which are targets of GFix. In total, GFix generates patches for 129 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 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 85 of the patches directly.

GFix decides not to fix the remaining 22 bugs for three reasons. For nine of the bugs, the blocking goroutine is the parent, and thus GCatch decides not to fix them. For ten of the bugs, there are side effects after the *o2* instruction, and if GFix unblocks the *o2*, the program semantics may be changed. The left three bugs involve one or more than two goroutines.

Performance. For 119 out of the 129 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. It is less than 4% for all bugs. Actually, there are only 16 bugs with overhead larger than 1%. The results show that GFix's patches incur a negligible performance impact.

Readability. We measure the readability of GFix's patches by counting the changed lines of source code. On average, GFix changes 2.7 lines of code to fix a bug. There are 103 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 five 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.

## 6. Discussion and Future Work

Go advocates the use of channel for thread communication and also provides several new libraries and mechanisms for concurrent programming. Programmers choose Go largely due to its concurrency features. However, GCatch shows that programmers make many mistakes when using the concurrency features like channel and `select`. Fortunately, GFix demonstrates that those related concurrency bugs can be fixed by leveraging the concurrency features. One lesson we learned is that bug detection and bug fixing for a new programming language should center around that language's unique features.

***Limitations and Future Work of GCatch.*** The constraint system in GCatch models only channel and mutex, and thus GCatch can only detect BMOC bugs caused by them. However, after changing other primitives to channels (as we did for mutex), GCatch can detect blocking bugs caused by these primitives as well. Misuse of channel can also cause non-blocking bugs. For example, closing an already closed channel triggers a panic. Another direction to extend GCatch is to model and detect when non-blocking misuse-of-channel bugs happen.

GCatch does generate some false positives. Many of these are due to the limitations of the static analysis packages we used (*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 relationships 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 bugs matching 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 in many other new languages (*e.g.*, Rust, Clojure), and thus our techniques can potentially be applied to those languages after some implementation changes to fit the languages. Our experience in extending existing constraint systems by modeling channel-related concurrency features in Go motivates future researchers to enhance existing techniques by handling unique features of new programming languages.

## 7. Related Work

***Static Concurrency Bug Detection.*** Many research works have been conducted to statically detect deadlocks [35, 44, 65, 71] and data races [40, 41, 57, 58, 64, 66, 76] 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 channel. Fahndrich et al. [26] designed 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 [27, 46, 47, 59, 69]. 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 [14], and the built-in vet tool [4]. Each contains four concurrency-bug detectors. Although some of the detectors overlap with our traditional detectors in Section 3.5, our detectors conduct more complex analysis and can detect more bugs. For example, staticcheck's double-lock detector only identifies cases where deferring a locking operation is right after another locking operation on the same mutex, and it detects none of the double locks reported by GCatch.

***Dynamic Concurrency Bug Detection.*** There are many dynamic techniques for detecting concurrency bugs, like data races [23, 29, 30, 31, 32, 42, 50, 68, 70, 81, 82], atomicity violations [20, 24, 54, 60, 73, 77], order violations [28, 55, 80, 83] and deadlocks [18, 19, 25, 72], that occur in traditional programming languages (*e.g.*, C/C++, Java). Go also provides two built-in dynamic detectors for deadlocks and data races, respectively. However, 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, so that 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 [39, 78, 79, 84]. There are also techniques that fix non-blocking bugs, and they achieve their goal by disabling bad timing of accessing a shared resource [37, 38, 49, 52] or by eliminating the sharing altogether [33]. 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 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.

## References

[1] A curated list of awesome Go frameworks, libraries and software. https://github.com/avelino/awesome-go.

[2] A golang ebook intro how to build a web with golang. https://github.com/astaxie/build-web-application-with-golang.

[3] A high performance, open-source universal RPC framework. https://github.com/grpc/grpc-go.

[4] Command vet. URL: https://golang.org/cmd/vet/.

[5] Data Race Detector. https://golang.org/doc/articles/race_detector.html.

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

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

[8] Package AST. https://golang.org/pkg/go/ast/.

[9] Package CHA. https://godoc.org/golang.org/x/tools/go/callgraph/cha.

[10] Package Deadlock. https://godoc.org/github.com/sasha-s/go-deadlock.

[11] Package Pointer. https://godoc.org/golang.org/x/tools/go/pointer.

[12] Package SSA. https://godoc.org/golang.org/x/tools/go/ssa.

[13] Production-Grade Container Orchestration. https://kubernetes.io/.

[14] Staticcheck – a collection of static analysis tools for working with Go code. URL: https://github.com/dominikh/go-tools.

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

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

[17] Macy Bayern. Top 10 programming languages developers want to learn in 2019. https://www.techrepublic.com/article/top-10-programming-languages-developers-want-to-learn-in-2019/.

[18] Yan Cai and W. K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland, June 2012.

[19] Yan Cai, Shangru Wu, and W. K. Chan. Conlock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, Hyderabad, India, May 2014.

[20] Lee Chew and David Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, New York, NY, USA, 2010.

[21] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check flash protocol code. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00)*, New York, NY, USA, 2000.

[22] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, Berlin, Heidelberg, 2008.

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

[24] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, New York, NY, USA, 2004.

[25] Vojtundefinedch Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. Precise predictive analysis for discovering communication deadlocks in mpi programs. *ACM Transactions on Programming Languages and Systems*, 2017.

[26] Manuel Fähndrich, Sriram Rajamani, and Jakob Rehof. Static deadlock prevention in dynamically configured communication networks. 2008.

[27] Julia Gabet and Nobuko Yoshida. Static race detection and mutex safety and liveness for go programs. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP '20)*, Berlin, Germany, 2020.

[28] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndstrike: Toward manifesting hidden concurrency typestate bugs. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, New York, NY, USA, 2011.

[29] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*, PLDI '15, New York, NY, USA, 2015.

[30] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, New York, NY, USA, 2014.

[31] Jeff Huang and Arun K. Rajagopalan. Precise and maximal race detection from incomplete traces. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*, New York, NY, USA, 2016.

[32] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*, New York, NY, USA, 2011.

[33] Jeff Huang and Charles Zhang. Execution privatization for scheduler-oblivious concurrent programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '2012)*, Tucson, Arizona, USA, October 2012.

[34] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, Seattle, Washington, USA, 2013.

[35] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, Lincoln, Nebraska, USA, November 2015.

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

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

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

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

[40] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE '09)*, New York, NY, USA, 2009.

[41] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices*, 2012.

[42] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: crowdsourced data race detection. In *Proceedings of the 24th ACM symposium on operating systems principles (SOSP '2013)*, 2013.

[43] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA '15)*, Baltimore, MD, USA, 2015.

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

[45] Sugandha Lahoti. Why Golang is the fastest growing language on GitHub. `https://hub.packtpub.com/why-golan-is-the-fastest-growing-language-on-github/8`.

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

[47] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*, New York, NY, USA, 2018.

[48] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 1993.

[49] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. Dfix: Automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '2019)*, Phoenix, AZ, USA, June 2019.

[50] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, New York, NY, USA, 2019.

[51] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '05)*, New York, NY, USA, 2005.

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

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

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

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

[56] Nuno Machado, Brandon Lucia, and Luís Rodrigues. Concurrency debugging with differential schedule projections. *ACM SIGPLAN Notices*, 2015.

[57] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, New York, NY, USA, 2007.

[58] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, New York, NY, USA, 2006.

[59] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*, New York, NY, USA, 2016.

[60] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, New York, NY, USA, 2009.

[61] Keval Patel. Why should you learn Go? `https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65`.

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

[63] Kevin Poulsen. Software Bug Contributed to Blackout. URL: https://www.securityfocus.com/news/8016.

[64] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, New York, NY, USA, 2006.

[65] Dawson R. Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In

*Proceedings of the 19th ACM symposium on Operating systems principles (SOSP '03)*, Bolton Landing, New York, USA, October 2003.

[66] Cosmin Radoi and Danny Dig. Practical static race detection for java parallel loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA '2013)*, New York, NY, USA, 2013.

[67] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '2014)*, Hong Kong, China, November 2014.

[68] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. 1997.

[69] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, New York, NY, USA, 2019.

[70] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, New York, NY, USA, 2008.

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

[72] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. A sound reduction of persistent-sets for deadlock detection in mpi applications. In *Proceedings of the 15th Brazilian conference on Formal Methods: foundations and applications (SBMF '12)*, Natal, Brazil, 2012.

[73] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*, New York, NY, USA, 2010.

[74] Minh-Phuc Tran. Use Go Channels as Promises and Async/Await. https://levelup.gitconnected.com/use-go-channels-as-promises-and-async-await-ee62d93078ec/.

[75] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, New York, NY, USA, 2019.

[76] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (FSE '07)*, New York, NY, USA, 2007.

[77] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342, Berlin, Heidelberg, 2010.

[78] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, Berkeley, CA, USA, 2008.

[79] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, New York, NY, USA, 2009.

[80] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, New York, NY, USA, 2009.

[81] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*, New York, NY, USA, 2005.

[82] Tong Zhang, Changhee Jung, and Dongyoon Lee. Prorace: Practical data race detection for production use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, New York, NY, USA, 2017.

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

[84] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. Undead: Detecting and preventing deadlocks in production software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*, Urbana-Champaign, IL, USA, 2017.