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

ACM Reference Format:
Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems.

ASPLOS '21, April 19–23, 2021, Virtual, MI, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446756

In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, MI, USA*. ACM, New York, NY, USA, 14 pages. 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.

```
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.

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

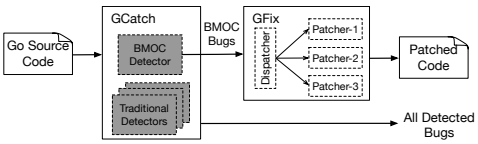

Figure 2: The workflow of GCatch and GFix.

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

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 (`Mutex`), read/write lock (`RWMutex`), 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_graph ← BuildCallGraph(P)
 3:     Alias ← ConductAliasAnalysis(C_graph)
 4:     Primitives ← SearchSynPrimitives(C_graph)
 5:     OP_map ← SearchSynOperations(Primitives, C_graph, Alias)
 6:     /*more details of lines 7 and 9 are in Section 3.2*/
 7:     D_graph ← BuildDependGraph(OP_map, C_graph)
 8:     for each channel c in Primitives do
 9:         scope, P_set ← DisentanglingAnalysis(c, OP_map, D_graph)
10:         GO_set ← SearchGoroutines(c, scope)
11:         /*more details of line 12 are in Section 3.3*/
12:         PCs ← ComputePathCombination(GO_set, P_set, OP_map, scope)
13:         for each path combination pc in PCs do
14:             /*more details of lines 15 – 23 are in Section 3.4*/
15:             Groups ← ComputeSuspiciousOpGroups(pc, P_set, GO_set)
16:             for each group (g) in Groups do
17:                 Φ_R ← ReachConstraint(g, P_set)
18:                 Φ_B ← BlockConstraint(g)
19:                 Φ ← Φ_R ∧ Φ_B
20:                 if Z3 finds a solution (s) for Φ then
21:                     ReportBug(g, s)
22:                 end if
23:             end for
24:         end for
25:     end for
26: 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_{set}) in a small analysis scope (*scope*). We will discuss the disentangling policy in Section 3.2.

Third, GCatch determines the goroutine set (GO_{set}) 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 GO_{set} (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 (Φ_R) for the program to execute just before the group operations and the constraints (Φ_B) for all group operations to block. If Z3 finds a solution for the conjunction of Φ_R and Φ_B 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 (*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_{set} at line 9) 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 identified functions and consider *scope* of c as the union of the computed scopes.

***How to compute** P_{set}?* GCatch builds a graph (D_{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_{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.

GCatch computes P_{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_{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_{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 GO_{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_{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 GO_{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_{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 (Φ_{R}) and whether all group operations can block (Φ_{B}) (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_{s7} indicates that before the sending operation at line 7 in Figure 1, CB_{s7} 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 Φ_R? Φ_{R} represents all the necessary constraints for goroutines in GO_{set} to execute just before operations in a suspicious group. Besides variables for channel states, Φ_{R} 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 $P_{(\text{si, rj})} = 1$, then the sending operation at line i and the receiving operation at line j execute at the same time ($O_{\text{si}} = O_{\text{rj}}$). For example, given the code snippet in Figure 1, $P_{(\text{s7, r10})} = 1$ means the sending operation at line 7 unblocks the receiving operation at line 10, and these two operations have the same execution order ($O_{\text{s8}} = O_{\text{r10}}$).

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

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

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:

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

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.

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 Φ_B? Similarly, Φ_B for a suspicious group is constructed by a conjunction of two sub-formulae: 1) Φ_{order} requires that instructions before the group operations must have smaller O variable values, compared with the group operations, and 2) Φ_{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_{set}, 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 Φ_R, Φ_{order} is "$O_3 < \ldots < O_{13} < O_{14} \wedge O_6 < O_7$" which is enforced by the two paths in the path combination, Φ_{spawn} is "$O_5 < O_6$" required by the goroutine creation at line 5, and Φ_{sync} is empty, since there is no synchronization operation belonging to any primitive in the P_{set} on the path of the parent goroutine and on the path of the child goroutine before line 7.

For Φ_B, Φ_{order} is "$O_3 < O_7 \wedge \ldots \wedge O_{14} < O_7 \wedge O_6 < O_7$", which means instructions before the instruction in the suspicious group (the sending at line 7) execute earlier. Φ_{sync} is "$CB_{s7} = BS$", 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 Φ_R and Φ_B. The solution is "$O_3 = 0 \wedge \ldots \wedge O_{14} = 4 \wedge O_6 = 5 \wedge O_7 = 6 \wedge CB_{s7} = 0$", which means when the program executes in the order $3 \rightarrow \ldots \rightarrow 14 \rightarrow 6 \rightarrow 7$, the child blocks at sending operation at line 7 forever.

3.5 Traditional Checkers

GCatch contains five additional checkers to detect three types of Go concurrency bugs. These checkers leverage old ideas that are effective at detecting concurrency bugs in traditional programming languages. We include the five checkers into GCatch to demonstrate traditional concurrency bugs are widespread in Go programs and to increase the bug coverage of GCatch. We briefly discuss the five checkers as follows.

GCatch detects traditional deadlocks caused by misuse of mutexes 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 [74, 75], 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 [29, 32, 50, 63]. GCatch detects concurrency bugs caused by errors when using the `testing` package. Each unit testing function in Go takes a `testing.T` object as a parameter. The `Fatal()` method and many other methods (*e.g.*, `FailNow()`, `Fatalf()`) of a `testing.T` object can only be called by the main goroutine running the testing function. GCatch checks whether a `Fatal()` call site is executed by a child goroutine and reports a bug if so.

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 language 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. Thus, the patches have good *readability* and are easily 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*. We will use the parent (child) goroutine and *Go-A* (*Go-B*) exchangeably 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*.

```
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.

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 *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

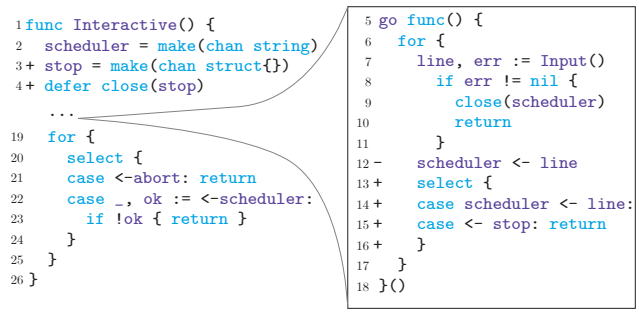

Figure 4: A multiple-operations bug in Go-Ethereum.

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 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 *o1*s 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 *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 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 `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.

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, 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.)

	GCatch								GFix			
App Name	**BMOC_C**	**BMOC_M**	**Forget Unlock**	**Double Lock**	**Conflict Lock**	**Struct Field**	**Fatal**	**Total**	**S.-I**	**S.-II**	**S.-III**	**Total**
Go	21_2	1_1	8_3	0_2	1_0	2_5	3_0	36_{13}	12	-	2	14
Kubernetes	14_5	1_0	1_0	1_0	-	5_6	10_0	32_{11}	8	-	-	8
Docker	49_8	-	1_1	2_3	1_0	3_1	-	56_{13}	40	1	6	47
HUGO	-	-	2_0	0_1	-	2_1	-	4_2	-	-	-	-
Gin	-	-	-	-	-	-	-	-	-	-	-	-
frp	-	-	1_0	-	-	-	-	1_0	-	-	-	-
Gogs	-	-	-	-	-	-	-	-	-	-	-	-
Syncthing	0_1	-	3_1	-	-	1_2	-	4_4	-	-	-	-
etcd	39_8	-	6_1	1_2	0_1	7_2	4_0	57_{14}	24	1	9	34
v2ray-core	-	0_1	-	2_1	2_1	3_0	-	7_3	-	-	-	-
Prometheus	2_1	-	1_1	1_1	0_2	0_2	-	4_7	2	-	-	2
fzf	-	-	0_1	-	-	-	-	0_1	-	-	-	-
traefik	-	-	-	-	-	-	-	-	-	-	-	-
Caddy	-	-	-	-	-	-	-	-	-	-	-	
Go-Ethereum	9_{19}	0_3	4_1	9_1	-	6_7	3_0	31_{31}	6	-	2	8
Beego	-	-	-	-	-	3_0	-	3_0	-	-	-	-
mkcert	-	-	-	-	-	-	-	-	-	-	-	-
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	1_0	1_0	2_0	10_1	4	-	1	5
bbolt	2_0	-	-	-	-	-	4_0	6_0	1	-	1	2
Total	147_{46}	2_5	32_{15}	19_{16}	9_5	33_{31}	26_0	268_{118}	99	4	21	124

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 [2] and another is an ebook on Go [4]. 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 [87]. 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.

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 the 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.

Results of the BMOC detector. The BMOC detector finds 149 new bugs from ten different applications. Among them, only two are caused by misuse of channels and mutexes (column "BMOC_M" in Table 1), while all the others involve channels only (column

"BMOC$_C$"). The large number of identified bugs demonstrates the effectiveness of the BMOC detector. In most of the detected BMOC bugs, a child goroutine is blocked forever and system resources 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 [87]. 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 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 goroutine 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 generate 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 generation time for Docker is 151 seconds, and it is the second longest. Bbolt 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 relationships, 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 using 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 changing 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 to 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 direction 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 constraints 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 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 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 techniques 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 [34, 77, 87]. 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.

REFERENCES

[1] A command-line fuzzy finder. https://github.com/junegunn/fzf.
[2] A curated list of awesome Go frameworks, libraries and software. https://github.com/avelino/awesome-go.
[3] A distributed, reliable key-value store for the most critical data of a distributed system. https://github.com/coreos/etcd.
[4] A golang ebook intro how to build a web with golang. https://github.com/astaxie/build-web-application-with-golang.
[5] A high performance, open-source universal RPC framework. https://github.com/grpc/grpc-go.
[6] A simple zero-config tool to make locally trusted development certificates with any names you'd like. https://github.com/FiloSottile/mkcert.
[7] Actor model. https://en.wikipedia.org/wiki/Actor_model#Contrast_with_other_models_of_message-passing_concurrency.
[8] CockroachDB is a cloud-native SQL database for building global, scalable cloud services that survive disasters. https://github.com/cockroachdb/cockroach.
[9] Command vet. URL: https://golang.org/cmd/vet/.
[10] Communicating sequential processes. https://en.wikipedia.org/wiki/Communicating_sequential_processes.
[11] Data Race Detector. https://golang.org/doc/articles/race_detector.html.
[12] Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/.
[13] Effective Go. https://golang.org/doc/effective_go.html.
[14] Go (programming language). https://en.wikipedia.org/wiki/Go_(programming_language).
[15] Mars Pathfinder. URL: https://en.wikipedia.org/wiki/Mars_Pathfinder.
[16] Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum.
[17] Package AST. https://golang.org/pkg/go/ast/.
[18] Package CHA. https://godoc.org/golang.org/x/tools/go/callgraph/cha.
[19] Package Deadlock. https://godoc.org/github.com/sasha-s/go-deadlock.
[20] Package Pointer. https://godoc.org/golang.org/x/tools/go/pointer.
[21] Package SSA. https://godoc.org/golang.org/x/tools/go/ssa.
[22] Production-Grade Container Orchestration. https://kubernetes.io/.
[23] Staticcheck — a collection of static analysis tools for working with Go code. URL: https://github.com/dominikh/go-tools.
[24] The Go Blog: Share Memory By Communicating. https://blog.golang.org/share-memory-by-communicating.
[25] Sameer Ajmani. Advanced Go Concurrency Patterns. https://talks.golang.org/2013/advconc.slide.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] Randy Coulman. Debugging Race Conditions and Deadlocks. URL: https://randycoulman.com/blog/2013/03/05/debugging-race-conditions-and-deadlocks/.
[31] Russ Cox. Bell Labs and CSP Threads. http://swtch.com/~rsc/thread/.
[32] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, New York, NY, USA, 2013.
[33] 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.
[34] Nicolas Dilley and Julien Lange. An empirical study of messaging passing concurrency in go projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*, 2019.
[35] 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.
[36] Manuel Fähndrich, Sriram Rajamani, and Jakob Rehof. Static deadlock prevention in dynamically configured communication networks. 2008.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. A graph based approach for mpi deadlock detection. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, Yorktown Heights, NY, USA, 2009.
[42] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. Mpi runtime error detection with must: Advances in deadlock detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, Salt Lake City, Utah, 2012.
[43] 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.
[44] 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.
[45] 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.
[46] 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.
[47] 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.
[48] 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.
[49] 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.
[50] 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.
[51] 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.
[52] 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.
[53] 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.
[54] 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.
[55] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. *ACM SIGPLAN Notices*, 2012.
[56] 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.
[57] 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.
[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Kedar S. Namjoshi. Are concurrent programs that are easier to write also easier to check? 2008.

[70] 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.

[71] 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.

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

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] Paul Rubens. Why Software Testing Can't Save You From IT Disasters. URL: https://www.cio.com/article/2378046/net/why-software-testing-can-t-save-you-from-it-disasters.html.

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

[80] 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.

[81] 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.

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

[83] 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.

[84] 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.

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

[86] Deepti Tripathi. Why Is Golang so Popular These Days? https://dzone.com/articles/why-is-golang-so-popular-these-days-here-are-your.

[87] 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.

[88] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. Isp: A tool for model checking mpi programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, Salt Lake City, UT, USA, 2008.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.