1. Motivation

Go is a statically typed programming language designed by Google in 2009 [9]. In recent years, Go has gained increasing popularity in building software in production environments. These Go programs range from libraries [3] and command-line tools [1, 4] to systems software, including container systems [8, 15], databases [2, 5], and blockchain systems [11].

The major design goal of Go is to provide an efficient and safe way for developers to write concurrent programs [10]. To achieve this purpose, it provides lightweight threads (called goroutines) that can be easily created, and 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 [16, 17, 40]. In addition, Go also provides several unique primitives and libraries for concurrent programming.

Unfortunately, there are still many concurrency bugs in Go, the type of bugs that are most difficult to debug [36, 37] and severely hurt the reliability of multi-threaded software systems [33, 43]. Ray et al. [45] compared multiple programming languages and found that Go is especially prone to concurrency bugs. Moreover, Tu et al. [49] reported 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. Thus, it is urgent to combat concurrency bugs in Go, especially those caused by misuse of channels, since Go advocates the use of channel and many programmers choose Go because of this very feature [30, 42].

2. Limitations of the State of the Art

Many advanced techniques have been built for concurrency bug detection and automated concurrency bug fixing. Unfortunately, none of them are effective at tackling channel-related bugs for Go.

Existing detection techniques fail to identify channel-related bugs in large Go software systems for three reasons. First, concurrency bug detection techniques designed for classic programming languages [19, 20, 24, 29, 38, 39, 44, 47, 48] mainly rely on analyzing shared-memory accesses, shared-memory primitives, or channels different from those used in Go (i.e., they are based on a different design model [20, 48]) or have different channel operations [19]). Thus, these techniques cannot detect bugs caused by misuse of channels in Go. Second, the three concurrency bug detectors released by the Go team [6, 7, 13] cover only limited buggy code patterns and cannot identify the majority of Go concurrency bugs in the real world [49]. Third, although recent techniques can identify blocking bugs in Go using model checking [21, 31, 32, 41, 46], 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).

Effective techniques have been proposed to fix concurrency bugs due to misuse of shared-memory concurrency [25, 26, 34, 35] and prevent lock-related deadlocks at runtime [27, 51, 52, 53]. Although effective, fixing channel-related bugs in Go requires different strategies — disabling bad timing of accessing a shared resource, changing lock acquisition orders, or adding coarse-granularity locks usually does not help fix channel-related bugs. Moreover, Go provides many concurrency features (e.g., channel, select) that are frequently used by Go programmers. Leveraging those features could potentially generate patches with good readability, since those patches would be similar to what developers usually do during programming and bug fixing. Unfortunately, existing concurrency bug fixing techniques do not exploit these new concurrency features.

3. Key Insights

We believe both concurrency bug detection and concurrency bug fixing for Go should center around Go’s channel-related concurrency features.

With respect to detection, we anticipate that many developers choose Go because of its channel-related concurrency features. Unfortunately, developers are generally trained more to program shared-memory concurrency than to program message-passing concurrency, and thus they are more likely to make mistakes when using channels, causing channel-related bugs. A promising way of detecting these bugs is to extend existing constraint systems by modeling channel operations, since constraint solving has successfully been used to combat concurrency bugs due to misuse of shared memory.

One challenge of automated bug fixing is improving the readability for generated patches, so that they will be more readily accepted by developers. One potential solution for Go concurrency bugs is to leverage channel-related features. Since those features are powerful and already frequently used by developers, using them aligns with programmers’ usual practice and can reduce the lines of changed code for generated patches. Thus, it will be easier for developers to validate and
accept generated patches.

4. Main Artifacts

In this paper, we build a static concurrency bug detection system, GCatch, and an automated concurrency bug fixing system, GFix (see Figure 2 in the main paper). GCatch focuses on detecting blocking misuse-of-channel (BMOC) bugs, since the majority of channel-related bugs in Go are blocking bugs [49]. It also contains five additional detectors based on effective approaches for discovering concurrency bugs in classic programming languages.

The innovation of GCatch lies in applying constraint solving to identify BMOC bugs in large Go systems software. Its design takes two steps. To scale to large Go software, GCatch conducts reachability analysis to compute the relationship between synchronization primitives of an input program, and leverages that relationship to disentangle the primitives into small groups. GCatch inspects each group only in a small program scope. To identify BMOC bugs, GCatch enumerates execution paths for all goroutines executed in a small scope, uses a novel constraint system to precisely describe how channel operations proceed and block, and invokes Z3 [18] to search for a possible execution causing some operations to block forever (i.e., a blocking bug). Existing constraint systems model primitives (e.g., mutex) without states [22, 23, 28, 50]; however, since a channel's behavior depends on its states (e.g., how many elements are in the channel), modeling channels is much more complex.

Once GCatch has detected BMOC bugs, GFix leverages channel-related concurrency features to generate patches for those bugs, and the patches have good performance and readability. GFix conducts static analysis to categorize input BMOC bugs into three groups and provides different strategies for each. GFix automatically increases channel buffer sizes or uses keywords defer and select to change blocking channel operations to be non-blocking and fix bugs in each group. Since GFix's patches change only the blocking channel operations without influencing other parts of the programs, the patches have little performance impact. Unlike existing bug fixing techniques, GFix's patches mimic the way developers usually program Go in reality and change only a few lines of code. Thus, the patches are easy for developers to validate.

We implemented GCatch and GFix using the SSA package [14] and the AST package [12]. GCatch detects BMOC bugs in Go by modeling channel-related concurrency features, while GFix fixes BMOC bugs using channel-related language features. The two techniques constitute an end-to-end system to combat BMOC bugs and improve the reliability of production-run Go systems software.

5. Key Results and Contributions

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 119 previously unknown traditional bugs; the number of false positives reported amounts to less than half the number of real bugs. We reported all detected bugs to developers. So far, 210 bugs (125 BMOC bugs and 85 traditional 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 inspecting it in 25.6 hours and find 15 BMOC bugs, demonstrating its capability to analyze large Go software. Overall, GCatch can effectively detect BMOC bugs in large, real Go software.

GFix generates patches for 124 detected BMOC bugs. All of them are correct, and almost all of them incur less than 1% runtime overhead. On average, each patch changes 2.67 lines of code, and in 99 of the patches, only one line of code is changed. So far, 87 of the generated patches have been applied directly by developers. In summary, GFix's patches have good performance and readability, and are easily validated and accepted by developers.

In summary, we make the following contributions:

- Based on a novel constraint system for channel operations and an effective disentangling policy, we build a concurrency bug detection system that can analyze large Go systems software.
- We design an automated bug fixing system for BMOC bugs in Go. This system generates correct patches with good performance and readability.
- We conduct thorough experiments to evaluate our systems. We identify and patch hundreds of previously unknown concurrency bugs in real Go software.
- Go covers many concurrency features in many other new programming languages (e.g., Rust, Kotlin), and thus our techniques can potentially be applied to those languages. Our experience in extending existing constraint systems by modeling a new concurrency primitive with states motivates future researchers to enhance existing techniques by handling unique features of new programming languages.

6. Why ASPLOS

This paper describes a concurrency bug detection technique and a concurrency bug fixing technique for the new programming language Go. Both of the detection technique and the fixing technique are based on static program analysis. Moreover, the detection leverages a novel constraint system, belonging to the formal methods area. Thus, this paper is relevant to the area of programming languages.

Go is widely used to build concurrent systems. The ultimate goal of this paper is to improve the reliability of Go systems software. In particular, we evaluate our techniques on two famous container systems (Docker and Kubernetes). Therefore, this paper is relevant to the area of operating systems.
References


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


