Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems
Extended Abstract

Ziheng Liu[1], Shuofei Zhu[1], Boqin Qin[2], Hao Chen[3], Linhai Song[1]

[1]Pennsylvania State University, [2]BUPT, [3]University of California, Davis

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 channel 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 channel is even more likely to cause blocking bugs (*e.g.*, deadlock) than misuse of mutex. Thus, it is urgent to combat concurrency bugs in Go, especially those caused by misuse of channel, 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 [19, 48] or have different channel operations [20]). Thus, these techniques cannot detect bugs caused by misuse of channel 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 concurrency 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 153 previously unknown BMOC bugs and 123 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, 199 bugs (124 BMOC bugs and 75 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 12 hours and find 17 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 129 detected BMOC bugs. All of them are correct, and almost all of them incur less than 1% runtime overhead. On average, each patch changes 2.7 lines of code, and in 103 of the patches, only one line of code is changed. So far, 85 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, Clojure), 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 emphasizes the synergy of programming languages and operating systems.

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

[1] A command-line fuzzy finder. https://github.com/junegunn/fzf.

[2] A distributed, reliable key-value store for the most critical data of a distributed system. https://github.com/coreos/etcd.

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

[4] A simple zero-config tool to make locally trusted development certificates with any names you'd like. https://github.com/FiloSottile/mkcert.

[5] CockroachDB is a cloud-native SQL database for building global, scalable cloud services that survive disasters. https://github.com/cockroachdb/cockroach.

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

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

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

[9] Effective Go. https://golang.org/doc/effective_go.html.

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

[11] Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum.

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

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

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

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

[16] The Go Blog: Share Memory By Communicating. https://blog.golang.org/share-memory-by-communicating.

[17] Russ Cox. Bell Labs and CSP Threads. http://swtch.com/~rsc/thread/.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

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

[38] 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.

[39] 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.

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

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.