Who Goes First? Detecting Go Concurrency Bugs via Message Reordering

Ziheng Liu
Pennsylvania State University
USA

Shihao Xia*
Pennsylvania State University
USA

Yu Liang
Pennsylvania State University
USA

Linhai Song
Pennsylvania State University
USA

Hong Hu
Pennsylvania State University
USA

ABSTRACT

Go is a young programming language invented to build safe and efficient concurrent programs. It provides goroutines as lightweight threads and channels for inter-goroutine communication. Programmers are encouraged to explicitly pass messages through channels to connect goroutines, with the purpose of reducing the chance of making programming mistakes and introducing concurrency bugs. Go is one of the most beloved programming languages and has already been used to build many critical infrastructure software systems in the data-center environment. However, a recent study shows that channel-related concurrency bugs are still common in Go programs, severely hurting the reliability of Go applications.

This paper presents GFuzz, a dynamic detector that can effectively pinpoint channel-related concurrency bugs by mutating the processing orders of concurrent messages. We build GFuzz in three steps. We first adopt an effective approach to identify concurrent messages and transform a program to process those messages in any given order. We then take a fuzzing approach to generate new processing orders by mutating exercised ones and rely on execution feedback to prioritize orders close to triggering bugs. Finally, we design a runtime sanitizer to capture triggered bugs that are missed by the Go runtime. We evaluate GFuzz on seven popular Go software systems, including Docker, Kubernetes, and gRPC. GFuzz finds 184 previously unknown bugs and reports a negligible number of false positives. Programmers have already confirmed 124 reports as real bugs and fixed 67 of them based on our reporting. A careful inspection of the detected concurrency bugs from gRPC shows the effectiveness of each component of GFuzz and confirms the components' rationality.

CCS CONCEPTS

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

*Shihao Xia contributed equally with Ziheng Liu in this work.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507753

KEYWORDS

Go; Concurrency Bugs; Bug Detection; Dynamic Analysis; Fuzzing

ACM Reference Format:
Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. 2022. Who Goes First? Detecting Go Concurrency Bugs via Message Reordering. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507753

1 INTRODUCTION

Go is an industrial programming language invented by Google, meant for building safe and efficient concurrent programs [71]. In recent years, Go's popularity has increased dramatically, making it one of the most beloved and one of the most wanted programming languages [52]. Programmers have already adopted Go to build important infrastructure software, such as Docker [10], Kubernetes [35], and gRPC [19].

To facilitate multi-threaded programming, Go offers several built-in features, like goroutines and channels. A goroutine is a lightweight thread that can be created and reused efficiently in the user space, while a channel is a message-passing primitive for communication between different goroutines. Go programmers are encouraged to use channels instead of shared memory to connect concurrent goroutines, as explicit message passing is commonly believed to be more resistant to concurrency bugs [14, 21, 63].

Unfortunately, concurrent Go programs are still difficult for programmers to think through and program correctly. Concurrency bugs, which are notoriously difficult to debug [7, 33], are still common in Go programs [56, 66]. A recent empirical study counterintuitively reveals that incorrect message passing contributes a significant proportion of Go concurrency bugs, and misuse of channels can be more likely to cause such bugs under certain circumstances than misuse of traditional primitives (*e.g.*, mutexes) that protect shared-memory accesses [66].

Existing bug-detection techniques cannot effectively discover Go concurrency bugs, especially those due to wrong message passing. The main reason is that most existing methods are built for traditional programming languages (*e.g.*, C/C++, Java), and they only monitor shared-memory primitives and shared-memory accesses [2, 3, 11, 28–31, 34, 39, 39, 41, 48–50, 53, 54, 58]. They can neither handle message-passing primitives (*e.g.*, channels) nor detect any message-passing-related bugs. Existing model-checking techniques can systematically examine all possible message orders to pinpoint

```
1  // go parent()
2  func parent() {  // parent goroutine
3      ... // initialize object daemon
4      ch, errCh := daemon.discoveryWatcher.Watch()
5      select {
6      case <- Fire(1 * time.Second):
7          Log("Timeout!")
8      case e := <-ch:
9          if !reflect.DeepEqual(e, expected) {
10             Log("Unexpected!")
11         }
12     case e := <-errCh:
13         Log("Error!")
14     }
15     return
16 }
17 func (s *Discovery) Watch() (chan discovery.Entries, chan error) {
18 -   ch := make(chan discovery.Entries)
19 -   errCh := make(chan error)
20 +   ch := make(chan discovery.Entries, 1)
21 +   errCh := make(chan error, 1)
22     go func() { // child goroutine
23         entries, err := s.fetch()
24         if err != nil {
25             errCh <- err
26         } else {
27             ch <- entries
28         } ...
29     }()
30     return ch, errCh
31 }
```

Figure 1: A Blocking Bug in Docker and Its Patch. ***The code has been simplified for illustration purposes.***

bugs in distributed systems [38, 46, 72]. However, since only very few message orders can lead to concurrency bugs, exhaustively inspecting all message orders is not efficient to detect channel-related bugs in Go programs. We can also find several bug detectors built for Go. However, the static detectors either cover a limited number of buggy code patterns [15, 27], fail to scale to large, real Go programs [13, 36, 37, 51, 57], or report significant numbers of false positives and false negatives due to imprecise alias analysis [43]. The dynamic detectors for Go merely report concurrency bugs triggered in a given execution [6, 17, 20, 67]. Without the capability to alter programs' execution states and increase the chance of exposing bugs, these dynamic detectors miss many concurrency bugs [66].

Figure 1 shows a channel-related[1] concurrency bug from Docker. The parent goroutine calls an object method through an indirect call at line 4. The callee is actually function `Watch()` at lines 17–31. `Watch()` creates two unbuffered channels `ch` and `errCh` at lines 18 and 19 firstly, then starts a child goroutine at line 22, and finally returns the two channels at line 30. The child goroutine calls `s.fetch()` at line 23. It checks the return value and sends a message either to channel `errCh` at line 25 or to channel `ch` at line 27. Meanwhile, the parent goroutine blocks at the select statement at lines 5–14 until it either receives a message from `Fire()` after one second or from one of the two returned channels. If the message from `Fire()` comes first, the parent goroutine chooses the first `case`, which merely logs the timeout and returns. After that, no other goroutines have references to channel `ch` or channel `errCh`, and thus no goroutines can receive messages from the two channels anymore. Since both of the channels are unbuffered, the child goroutine blocks at one of the sending operations (line 25 or line 27) endlessly.

This example demonstrates the difficulty in detecting Go concurrency bugs related to channels. The bug only manifests when ① the message from `Fire()` at line 6 arrives earlier than the other two messages, so that the `select` proceeds with the first `case`. Meanwhile, the detector should be able to infer that ② no other goroutines have references to `ch` or `errCh` and can unblock the child goroutine. Static techniques cannot effectively infer the targets of indirect calls, like the one at line 4, and thus they cannot determine whether the child goroutine can be unblocked (condition ②). In offline testing, we notice that the message from `Fire()` never comes first and thus current dynamic methods will miss the bug due to the lack of condition ①.

In this paper, we propose a new dynamic-analysis tool, GFuzz, to effectively detect channel-related Go concurrency bugs. We consider both blocking bugs, where one or more goroutines are stuck in their executions, and non-blocking bugs, where all goroutines can finish but generate undesired results (*e.g.*, panics) [66]. GFuzz focuses on concurrent messages that are unique to the Go programming language. Since the processing order of these messages is non-deterministic by design, programmers must guarantee a Go program works well under all possible processing orders. However, due to the huge number of possible orders, programmers with limited time and energy are prone to missing some orders, which may bring in channel-related bugs.

Guided by this intuition, GFuzz intentionally mutates the order of concurrent messages to direct tested programs to different execution states and increases the chance of triggering both blocking bugs and non-blocking bugs. Meanwhile, it monitors program executions to capture triggered blocking bugs that are (largely) undetected by the Go runtime[2]. Take Figure 1 as an example. In theory, the message at line 6 could arrive before the ones at line 8 and line 12 *or* after them. No matter the order, the code should work in both scenarios. However, the programmer does not account for the first scenario, thereby introducing the aforementioned bug into this example. By intentionally mutating the message order, GFuzz can inspect both scenarios and detect the bug.

Although reordering concurrent messages is straightforward, we need to tackle three challenges to build a practical bug detector.

First, how to identify concurrent messages? Without precise information about concurrent messages, GFuzz may enforce a message order that conflicts with an existing happens-before relation, leading to a false deadlock that would never happen in real executions. To answer this question, we adopt a simple method that considers messages waited for by the same `select` statement. Channel operations under the same `select` can happen simultaneously and thus their messages are concurrent. Moreover, mutating the order of these messages preserves the program semantics and ensures the alteration of program execution (*i.e.*, different exercised `case`s). To enforce a given message order, GFuzz transforms each `select` in a way that a particular `case` is preferred over all others.

[1] We use "channel-related bug" and "message-passing-related bug" interchangeably, as most message-passing-related bugs are due to misuse of channels [43, 66].

[2] The runtime can capture triggered channel-related non-blocking bugs.

In addition, GFuzz uses a timeout mechanism to avoid false deadlocks. The mechanism falls back to the original execution if the preferred `case` is not chosen within a time threshold.

Second, how to identify and prioritize suspicious message orders? Since a given program could have a tremendous number of possible message orders, it is impractical to enumerate all of them. To address this challenge, we use the fuzzing method to generate new message orders from existing ones and rely on the execution feedback to prioritize interesting orders closer to triggering new bugs. We design several metrics to measure how a program conducts channel operations when following a particular message order, such as whether a new channel that has never been observed in historical executions is created, how many distinct consecutive execution pairs of channel operations there are, and whether a channel is fulfilled. We integrate all the metrics into a unified formula and use this formula to calculate a score for each order. With the scores, we identify and prompt interesting orders.

Third, how to identify a channel-related blocking bug? When one goroutine blocks at a channel operation, any other goroutines with references to the same channel can potentially unblock it in the future. Without careful examination of the execution state, it is easy to miss bugs or report false alarms. To tackle this issue, we design a runtime sanitizer that dynamically tracks the propagation of channel references among all goroutines. Based on the channel-goroutine relationship, we design a novel algorithm to identify blocking goroutines that cannot be unblocked by any others afterward and thus detect channel-related blocking bugs.

The novelty of GFuzz lies in ① tailoring (but not directly applying) existing ideas (*e.g.*, message reordering, feedback-guided fuzzing) for message-passing concurrency and Go programs and ② building a novel sanitizing algorithm to effectively pinpoint channel-related blocking bugs and complement the Go runtime's detection capability on channel-related non-blocking bugs. We envision that GFuzz can be used as an in-house testing tool. After launching a Go application with existing program inputs or unit tests, GFuzz would automatically explore various program execution states caused by different processing orders of concurrent messages and pinpoints previously unknown channel-related bugs.

To evaluate GFuzz, we take seven popular real-world Go software systems, including Docker, Kubernetes, and gRPC. In total, GFuzz finds 184 previously unknown bugs, including 170 blocking bugs and 14 non-blocking bugs, and reports 12 false positives. We have responsibly reported all the bugs. Thus far, programmers have confirmed 124 bugs and fixed 67 of them based on our reporting. In addition, we systematically compare GFuzz with the most recent static Go concurrency bug detector GCatch. The bugs reported by GFuzz in its first three hours of execution are significantly more than all the bugs pinpointed by GCatch, which confirms GFuzz indeed advances the state of the art of Go concurrency bug detection. Moreover, we carefully inspect concurrency bugs detected in gRPC by disabling each component of GFuzz. We observe that the full-featured GFuzz pinpoints most concurrency bugs, demonstrating the rationality of GFuzz's design.

Overall, we make the following three contributions.

- We tailor an existing approach, message reordering, for Go to proactively trigger concurrency bugs in Go programs.
- We design and implement GFuzz that can effectively detect channel-related Go concurrency bugs via order mutation, order prioritization and runtime detection.
- We conduct thorough experiments to evaluate GFuzz. GFuzz has detected 184 previously unknown channel-related bugs in real Go software.

All our code and experimental data can be found at *https://github.com/system-pclub/GFuzz*.

2 BACKGROUND

This section provides some necessary background information on this project, including concurrency mechanisms supported by the Go programming language, concurrency bugs in Go programs, and the problem scope of GFuzz.

2.1 Concurrency Mechanisms in Go

Go supports two methods (message passing and shared memory) for goroutine communication and synchronization. The language designers recommend message passing over shared memory, because they believe message passing is less likely to cause concurrency bugs [14, 21, 63]. In Go applications, channel (`chan`) is the primitive most often used to pass messages [66]. A goroutine can create, send to, receive from, and close a channel. Go supports both buffered channels and unbuffered channels (with a buffer size equal to 0). Whether a channel operation blocks depends on the number of available elements in the buffer and whether the channel is closed. For example, when a channel's buffer is empty, a goroutine that receives data from the channel blocks, until another goroutine sends data to the channel or closes it. If there are elements in the channel, a receiving operation returns immediately. Programmers must follow certain rules to code with channels to avoid concurrency bugs. For example, programmers must initialize a channel before using it and must not use a closed channel. This is because sending data to a `nil` channel blocks a goroutine endlessly and closing or sending data to a closed channel triggers a runtime panic.

Go allows a goroutine to wait for multiple channel operations using `select` statements. A `select` consists of several `case` statements, one for each channel operation, and an optional `default` clause. When none of the channel operations of a `select` is available, a goroutine either blocks at the `select` or continues its execution at the `default` (if it has one). If multiple channel operations are waited for by the same `select`, we intuit that they are concurrent and have no explicit happens-before relation. In Section 4, we leverage this intuition to identify and reorder concurrent messages.

In addition to relying on channels, Go allows multiple goroutines to communicate by accessing shared memory. Similar to traditional programming languages (*e.g.*, C/C++), Go provides several synchronization primitives to protect shared-memory accesses, including `Mutex`, `RWMutex`, `Cond`, `atomic`, and `WaitGroup`.

2.2 Concurrency Bugs in Go

Misuse of concurrency mechanisms may cause concurrency bugs in Go programs. Tu et al. [66] systematically studied Go concurrency issues by inspecting 171 historical bugs collected from six open-source Go applications, including Docker, Kubernetes, and gRPC. They categorized these bugs along two dimensions. First,

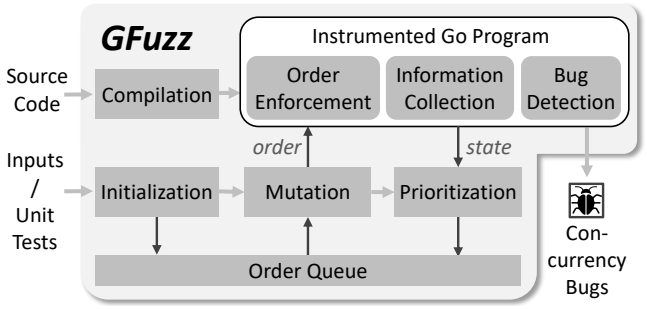

Figure 2: GFuzz System Overview. ***GFuzz takes a Go program and several program inputs or unit tests as inputs, and aims to detect concurrency bugs caused by misusing channel operations. It keeps generating new message orders, monitoring program execution states, and favoring interesting orders to speed up the bug-finding process.***

they divided the bugs into blocking bugs and non-blocking bugs, based on their symptoms. With blocking bugs, some goroutines are stuck and cannot complete their executions, while with non-blocking bugs, all goroutines can finish but may produce undesired results. Second, they referred to the bugs' underlying root causes and categorized them into those related to shared memory and those due to errors when passing messages.

For example, Figure 1 is a blocking bug and its root cause is the misuse of channel `ch` or channel `errCh`, while Docker#24007 is due to a runtime panic caused by closing an already closed channel, so that Docker#24007 is categorized as a channel-related non-blocking bug. On the other hand, traditional data races (*e.g.*, Kubernetes#9926, etcd#3576) are classified as shared-memory-related non-blocking bugs.

2.3 Problem Scope of GFuzz

In this paper, we follow the aforementioned two-dimensional categorization and design GFuzz to detect both channel-related blocking bugs and channel-related non-blocking bugs.

We define the problem scope like this for three reasons: ① bugs due to shared-memory misuses can be discovered by existing techniques built for traditional programming languages [3, 28–31, 34, 39, 48–50, 54, 61]; ② channels are the primitives most commonly used for message passing and most bugs caused by errors when passing messages are channel-related [66]; and ③ channel-related bugs can lead to severe consequences, including unexpected panics, program hangs, and resource leakages [42, 65, 70].

3 OVERVIEW

Figure 2 gives an overview of GFuzz, which takes a Go program and several program inputs or unit tests as input, and outputs detected concurrency bugs. GFuzz takes several steps to force a tested Go program to handle reordered concurrent messages for detecting channel-related concurrency bugs. For the sake of clarity, we use the term "program inputs" to represent both inputs of the whole program and unit tests.

In the first step, GFuzz compiles the program and uses the given inputs to initialize the order queue. Specifically, GFuzz automatically instruments the source code for three goals: to safely enforce particular message orders, to selectively collect runtime information, and to aggressively detect triggered concurrency bugs. After the compilation, GFuzz executes the program with each input. At this stage, GFuzz does not enforce any message order but merely records the order shown in the execution. Then, GFuzz adds the program input together with the observed order to the order queue. Such orders will be used as seeds to generate new orders.

In the second step, GFuzz sequentially fetches message orders from the queue and randomly mutates them to generate new orders. For each mutation result, GFuzz runs the program with the associated input and enforces the particular message processing order at runtime. If a message required by the order does not arrive within a predefined time window, GFuzz falls back to the program's original logic to avoid introducing false deadlocks. During the execution, GFuzz collects various runtime information to identify and prioritize interesting orders that are close to triggering bugs (Section 5). The runtime information reflects reached program states, like creating or closing channels. If one order triggers new program states, GFuzz will add it to the queue and also calculate a priority score for it to systematically distribute testing resources.

Last but not least, the runtime sanitizer continuously monitors the program execution and aggressively detects concurrency bugs at an early stage (Section 6).

GFuzz works like a traditional fuzzer, where both of them keep triggering different program states to detect bugs [22, 44, 76]. However, GFuzz is different from existing fuzzing tools in three aspects: ① GFuzz reorders concurrent messages to explore new program states, while fuzzing tools change program inputs; ② GFuzz determines whether a new program state is reached based on how channels behave, whereas fuzzing tools depend on branch information; and ③ GFuzz can effectively detect channel-related concurrency bugs, not just the program crashes caused by memory errors that are the targets of mainstream fuzzing tools [22, 44, 76].

4 REORDERING CONCURRENT MESSAGES

GFuzz reorders concurrent messages to explore program execution states and trigger channel-related bugs, rendering it fundamentally different from existing approaches that change memory-access orders or thread scheduling [39, 48, 53, 58]. Although detecting concurrency bugs by reordering messages is not a new idea [74], our contribution lies in tailoring the idea for Go programs. If two channel operations (sending or receiving) have no happens-before relation with each other, we consider them as concurrent operations and their processed messages as concurrent messages.

However, it is challenging to determine precisely whether two channel operations can execute simultaneously, since there are many synchronization operations in a Go program and it is difficult to analyze their interactions. GFuzz takes a simple, straightforward approach to pinpoint one particular, important type of concurrent channel operations. In Go, `select` enables one goroutine to wait for multiple channel operations at the same time. This semantics indicates that channel operations of the same `select` are concurrent. Therefore, GFuzz focuses on the channel operations of `select` statements. Since Go programs commonly use `selects` [9], GFuzz can pinpoint numerous concurrent channel operations. We will explore

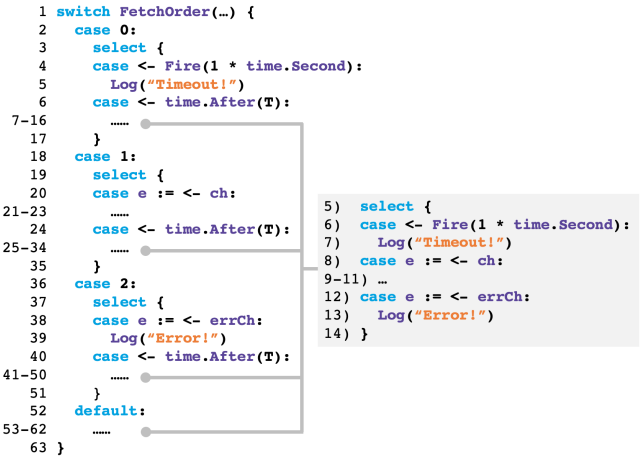

Figure 3: Order Enforcement Instrumentation for the `select` Statement in Figure 1.

advanced program analysis techniques to reveal more concurrent channel operations in future work.

Once we identify concurrent messages, GFuzz starts to mutate their orders to trigger different program states. Next, we introduce how GFuzz mutates existing message orders to generate new ones, and how GFuzz forces a tested program to follow a new order.

4.1 Mutating Message Orders

We first define the representation of message orders for effective mutations. Our observation is that among all `case` clauses within a `select`, the program picks only one to proceed at a time. It is the one whose associated message arrives earlier than all the other messages. Therefore, for each program execution, we use the sequence of picked `case` statements to represent the order of processed concurrent messages. To support effective order recording, we statically assign each `select` a unique ID, and allot a local index to each distinct `case` of a `select`. Now, we can represent a concrete message order with a sequence of tuples $[(s_0, c_0, e_0)...(s_n, c_n, e_n)]$, where s_i $(0 \leq i \leq n)$ represents a `select` ID, c_i represents the number of `case`s within the `select`, and e_i represents an exercised `case` index.

GFuzz randomly mutates an exercised order to generate new orders. Specifically, GFuzz goes through each tuple within the order and changes its `case` index to a random (but valid) value. GFuzz only changes exercised `case` clauses in a program run; it does not make any attempt to modify exercised `select` statements. The number of mutations generated for an order depends on runtime feedback when exercising the order, which we will discuss in Section 5.

Working example. Suppose the `select` at lines 5–14 in Figure 1 has ID 0; one program run goes over the `select` twice and chooses the second `case` for the two executions. Then, the message order of this run can be encoded as $[(0, 3, 1), (0, 3, 1)]$. GFuzz may produce nine possible orders, and one of them is $[(0, 3, 1), (0, 3, 2)]$.

4.2 Enforcing Message Order

To force a Go program to process concurrent messages in a given order, GFuzz conducts code transformation on all `select` statements in the program. Specifically, for a `select` with n cases, GFuzz replaces it with a `switch` with n cases and a `default` clause. The i-th `case` of the `switch` is to prioritize the i-th concurrent message waited for by the `select`, while the `default` clause is used when no order is specified for the `select`. Figure 3 shows part of the instrumented code for Figure 1. The original `select` has three `case` clauses, and thus the new `switch` has three `case`s.

To prioritize the i-th concurrent message, we implement the body of the i-th `switch case` as a `select` with two `case`s. One is the same as the i-th `select case`: the same channel operation and the same body. The other is a timer with period `T`, and the body is the same as the original `select`. When the execution reaches the i-th `switch case`, the i-th concurrent message is prioritized for selection within the `T` period. If the message does not arrive before the timeout, the execution falls back to the original `select`, and GFuzz leaves the program to choose any message. For example, in Figure 3 the `case 0` at line 2 is used to prioritize the message at line 6 in Figure 1, which is from `Fire()`. Therefore, the corresponding body (lines 3–17) is a `select` with two `case` clauses: the one at line 4 is for `Fire()`, while the other at line 6 waits for period `T`. If the message from `Fire()` does not arrive within `T`, the execution proceeds to execute lines 7–16, which is exactly the original `select` at lines 5–14 in Figure 1.

We design the timer (*e.g.*, line 6 in Figure 3) to avoid introducing false deadlocks (*i.e.*, deadlocks absent from real executions) to tested programs. Although a `select` does not require any happens-before relation between its `case` clauses, some `case`s may have such constraints due to other reasons. In this case, if we strictly require a `case` to proceed first (*i.e.*, we do not move on until the message arrives), the execution may hang there forever due to its conflict with an existing happens-before relation. By using the timer, we provide a fall-back mechanism to guarantee that the execution terminates and the message prioritization does not introduce any artificial blocking.

Since an order may contain many tuples, GFuzz uses function `FetchOrder()` (*e.g.*, line 1 in Figure 3) to make sure the specified tuple and the current `select` are consistent. Specifically, for each `switch`, `FetchOrder()` takes the ID of the replaced `select` as the argument, and returns the specified `case` index. Internally, `FetchOrder()` follows the input tuple order to separate tuples belonging to different `select`s into different arrays. In addition, `FetchOrder()` keeps an array index for each `select` to record the next tuple to be used for the `select`. If an input ID representing a `select` that does not appear in the input order, `FetchOrder()` returns -1 immediately so as to avoid forcing the `select` to prioritize any particular `case`. Otherwise, `FetchOrder()` sequentially uses tuples belonging to the `select` by incrementing the array index by one. If all tuples are used up, `FetchOrder()` changes the index value to zero and goes over the tuple array of the `select` again.

Working Example. The way that GFuzz transforms the `select` in Figure 1 is shown in Figure 3. Suppose the specified order is $[(0, 3, 0)]$. It aims to prioritize the first `case` of the original `select`, which is `Fire()`. In the execution, `FetchOrder()` returns a valid index 0, and the `switch` jumps to line 3 in Figure 3. If the `Fire()` message arrives before the timeout, the parent goroutine executes

Information	Semantics	Interesting Criteria	Identifier
`CountChOpPair`	# execs of each pair of channel operations	new pair / counter heavily changes	$(ID_{prev\_op} \gg 1) \oplus ID_{cur\_op}$
`CreateCh`	distinct channels created	new distinct channel created	ID of channel-create instruction
`CloseCh`	distinct channels closed	new distinct channel closed	ID of channel-create instruction
`NotCloseCh`	distinct channels remaining open	new distinct channel not closed	ID of channel-create instruction
`MaxChBufFull`	maximum fullness of each buffered channel	new maximum fullness	ID of channel-create instruction

Table 1: Runtime Information as Feedback. ***The interesting criteria determine whether we keep the current order for future mutations.***

line 5 and then returns from function `parent()`. In this way, we realize the prioritization of message `Fire()`.

5 FAVORING PROPITIOUS ORDERS

Due to the large amount of possible message orders, it is impractical to run tested programs under every order. Instead, we collect various types of runtime information to measure the quality of orders. We use these measurements as feedback to prioritize high-quality orders that exercise new program execution states and are close to triggering bugs. Of course, using runtime information to guide dynamic testing or fuzzing [5, 44, 76] is not a new approach. Our innovation lies in identifying which runtime information correlates with channel-related bugs and effectively leveraging that information.

5.1 Tracking Program Execution

To measure whether the current order triggers a new program state, GFuzz mainly tracks two types of information: ① interleavings of channel operations and ② channel states. Our observation is that channel-related concurrency bugs commonly occur when a program conducts an unexpected channel operation while the channel is in a particular state (*e.g.*, sending to a channel with a full buffer, closing an already closed channel). Unlike existing dynamic concurrency-bug detectors [39, 45, 53], GFuzz does not monitor shared-memory accesses. This is because doing so incurs a high runtime overhead and does not help pinpoint channel-related bugs.

Note that our collected information is an underapproximation of an execution's channel operations. While we could extend the information to cover more aspects, doing so would incur a heavier calculation cost and more runtime overhead while yielding only limited benefits.

Tracking Channel Operations. In theory, GFuzz could monitor channel operations for each goroutine, each channel, or the whole program. In our design, we choose to monitor the operations for each individual channel due to two reasons. First, if GFuzz monitors channel operations only within each individual goroutine, it will miss the orders of channel operations across different goroutines, which are the major sources of concurrency bugs. Second, if GFuzz monitors channel operations within a program as a whole, GFuzz has to maintain global data structures and enforce proper synchronization before accessing those data structures, which essentially sequentializes all channel operations. This will lead to a complicated design and even hide particular bugs. Therefore, monitoring channel operations for each channel is a reasonable angle to collect concurrency-related program states.

GFuzz takes two steps to track the execution of channel operations. First, GFuzz encodes the order of two same-channel operations. GFuzz's procedure resembles the way that fuzzing techniques encode the execution of two basic blocks [22, 44, 76]. Specifically, GFuzz assigns each channel operation (*e.g.*, initialization, sending) with a random ID and calculates the XOR of the IDs of two consecutive channel operations to represent the execution of the two operations. Since XOR is commutative, to differentiate operation *A* following operation *B* from *B* following *A*, GFuzz shifts the ID of the former operation one bit to the right before computing the XOR value. Second, GFuzz leverages a global data structure to record how many times each pair of channel operations has been executed. Specifically, GFuzz uses the XOR result as the offset to access the global structure and increments the content by one to indicate one execution. After each run, GFuzz leverages the content of the global structure to measure the quality of the current message order.

Table 1 shows the types of runtime information we collect to evaluate message orders. The first entry `CountChOpPair` indicates the number of executions of a pair of consecutive channel operations. In the global data structure, we allocate one two-byte value to each unique operation pair. The "Identifier" column shows how to calculate the identifier for each pair: ID_{prev_op} is the ID of the former operation; ID_{cur_op} is the ID of the latter operation; and we shift ID_{prev_op} one bit to the right and use the XOR result to represent the pair.

Tracking Channel States. As discussed in Section 2, if a channel becomes empty, full, or closed, the following operations on the channel may trigger channel-related bugs. Thus, GFuzz monitors channel states as the second type of feedback. Table 1 provides a list of interesting channel states we collect, including all channels created during each run (`CreateCh`), all channels closed (`CloseCh`), all channels remaining open (`NotCloseCh`), and the maximum fullness (*i.e.*, the maximum proportion of used slots) of each buffered channel (`MaxChBufFull`). To collect such information, we allocate a random ID to each channel-creation location and dynamically track the propagation of the channel. When a channel is created, closed, or fuller than before (*i.e.*, having fewer free slots in the buffer), we use the channel ID to access global data structures to record the event. We log all unclosed channels at the end of each execution.

5.2 Prompting Interesting Orders

GFuzz analyzes the recorded information for two purposes. First, after each execution, GFuzz leverages the information to determine whether the exercised order is interesting. If so, GFuzz adds the order to the order queue. Otherwise, GFuzz drops the order and

moves on to the next order for execution and measurement. Second, after adding an order to the order queue, GFuzz computes a score based on the recorded information to determine how many mutations to perform on the order.

Pinpointing Interesting Orders. The "Interesting Criteria" column of Table 1 shows our policy for identifying interesting orders. First, if the execution of an order triggers a new pair of consecutive channel operations, we treat the order as interesting. The term "new" here means that the operation pair has not been triggered in previous executions. In addition, we also treat an order as interesting if an operation pair's execution counter changes significantly from previous orders. Specifically, if the counter falls into a range $(2^{N-1}, 2^N]$ to which no previous counter belongs, then the order is considered interesting. Second, if the execution triggers a new channel operation, such as creating a new channel or closing (or not closing) a channel for the first time, we treat the current order as interesting. Third, if a buffered channel gets a larger maximum fullness (*i.e.*, the channel has fewer available slots), we also treat the current order as interesting. For example, if in previous runs buffered channel `ch` ever reaches at most 80% of its capacity, while in the current run its capacity reaches 90%, then we treat the current order as interesting. All interesting orders are put into the queue for further mutations.

Computing Mutation Numbers. Not all interesting orders are equal in triggering new bugs. Therefore, we calculate a priority score for each order using Equation 1 to distribute testing resources.

$$score = \sum \log_2 CountChOpPair + 10 * \#CreateCh + 10 * \#CloseCh + 10 * \sum MaxChBufFull \quad (1)$$

Specifically, we consider the sum of operation-pair counts, the number of distinct channels created, the number of distinct closed channels at program exit, and the sum of maximum channel fullness. We exclude the number of distinct not-closed channels, as the value has been covered by the number of channels created and the number of channels closed. The final score is a weighted sum of all factors. We choose this formula based on our intuition about how the quality of an order correlates with each factor. Our empirical evaluation shows that the scores calculated using this formula can help detect more concurrency bugs. We can tune the formula for different applications, but we leave this for future work.

Our testing process goes through the queue and picks up each order for mutation. An order's associated score indicates how many mutations we should perform on the order. We allocate more testing resources to mutate high-scoring orders and spend fewer resources on (or even skip) low-scoring ones. Specifically, if an order's score is `NewScore` and the previously observed maximum score is `MaxScore`, then the number of mutations generated for the order is the ceiling of "`NewScore`/`MaxScore` $*$ 5".

6 DETECTING CONCURRENCY BUGS

We design a novel runtime sanitizer to detect triggered channel-related bugs. Since the Go runtime can capture channel-related non-blocking bugs (sending to a closed channel and closing a closed channel), our sanitizer focuses on detecting channel-related blocking bugs.

However, timely detection of channel-related blocking bugs is difficult. This is because when a goroutine is waiting at a channel operation, any other goroutine holding a reference to the same channel can potentially unblock it by sending or receiving a message or closing the channel. Moreover, checking whether a goroutine can access a channel requires traversal of the goroutine's complex object reference graph, which is as time-consuming as garbage collection. Existing techniques ascertain that a blocking bug has occurred if there are unfinished goroutines when the main goroutine terminates [6, 67]. However, since a Go program can run for a long time, these techniques significantly delay their bug detection. Even worse, they may lose the execution information of triggered bugs, which is vital for bug diagnosis. To address these issues, we design our sanitizer to proactively track how channel references propagate among goroutines and maintain the relations between goroutines and channel references during execution.

Next, we first discuss important data structures used by the sanitizer and then explain how the sanitizer detects blocking bugs.

6.1 Data Structures

The sanitizer maintains three types of data structures in the Go runtime. GFuzz modifies both the Go runtime and the tested programs to update these structures. Our design hybridizes the modification because we can easily get some information from the application layer (*e.g.*, when a goroutine gains a reference to a channel), and reusing existing synchronization in the Go runtime helps to avoid introducing concurrency bugs.

`mapChToHChan` is a global variable that maps every application-layer channel (`chan`) to its internal representation (`hchan`) in the Go runtime. To update this data structure, we instrument each channel-creation site in the source code to call a newly added library function. The function takes a pointer to the created `chan` as input, and inserts an entry into `mapChToHChan` to map the `chan` to the latest `hchan` accessed by the current goroutine. We do not maintain such maps for other synchronization primitives (*e.g.*, mutexes), since Go represents them in the same way at the application layer and the runtime layer.

`stGoInfo` maintains information about goroutines. It tracks whether a goroutine blocks, and if so, for which primitive the goroutine is waiting. It also records which synchronization primitives (*e.g.*, `hchan`) a goroutine can access and which mutexes a goroutine has acquired. We allocate an `stGoInfo` object to each active goroutine.

To update `stGoInfo` objects, we modify both the Go runtime and the program source code to capture all goroutine-channel operations. In the Go runtime, we hook functions related to synchronization operations. For example, the Go runtime invokes function `makechan()` to handle channel creation operations. Therefore, we modify this function to record that the current goroutine refers to the created `hchan`. As another example, the Go runtime calls function `chansend()` to implement message sending. At the entry of this function, we record that the current goroutine is blocking and is waiting to send to the channel. Moreover, if the current goroutine's `stGoInfo` object does not contain the information that the goroutine has a reference to the channel, we update the `stGoInfo` object to record the information. At the exit of `chansend()`, we change the goroutine's state back to runnable.

```
1    go func() {
2  +   GainChRef(ch)
3  +   GainChRef(errCh)
4      entries, err := s.fetch()
5      ...
6    }
```

(a) application

```
7  + func GainChRef(c ch) {
8  +   h = mapChToHChan[c]
9  +   goInfo = currentGo.getStGoInfo()
10 +   goInfo.addChRef(h)
11 + }
```

(b) runtime

Figure 4: Instrumenting a Goroutine Creation for Bug Detection. ***Lines starting with + are added by GFuzz.***

We also instrument Go programs to make timely updates to `stGoInfo` objects when a goroutine gains or loses a reference to a primitive. Figure 4 shows an example. Lines 1 and 4–6 are the original code to create the child goroutine in Figure 1. GFuzz adds lines 2 and 3 to record that the child goroutine gains references to channels `ch` and `errCh`. Specifically, `GainChRef()` is a library function added by us. It first looks up `mapChToHChan` to find the corresponding `hchan` at line 8. It then retrieves the current goroutine's `stGoInfo` object at line 9. Finally, it records the new reference to `hchan` in the `stGoInfo` object at line 10.

`stPInfo` tracks information about synchronization primitives. We allocate one `stPInfo` object to each primitive to record which goroutines hold references to it. We mainly update `stPInfo` objects in Go runtime together with `stGoInfo` objects. We leverage existing synchronizations to guard concurrent accesses to `stPInfo` objects.

6.2 Detection Algorithm

Algorithm 1 shows how the sanitizer detects blocking bugs. Briefly, GFuzz first identifies a set of blocking goroutines whose `stGoInfo` objects indicate they are waiting for synchronization operations. Then it inspects their blocking conditions to determine whether they can be unblocked or not. For example, if a goroutine blocks at a channel operation while no goroutines holding a reference to the same channel are runnable now or in the future, then the blocking goroutine cannot be unblocked anymore, and thus we detect a blocking bug.

The detection algorithm takes a channel (`c`) and a goroutine (`g`) waiting for `c` as input. It uses the two sets created at line 2 to track visited primitives and goroutines, respectively. At line 3, the algorithm retrieves all goroutines related to channel `c` and puts them into a list (Go_{list}). The `stPInfo` object of channel `c` is used here to identify all goroutines that hold a reference to `c`.

The algorithm keeps executing the loop at lines 4–18 until it either finishes processing all the goroutines in Go_{list} and detects a bug or encounters a runnable goroutine, without detecting any bug. In each iteration, the algorithm first pops out a goroutine (`go`) from Go_{list} at line 5. If `go` is not blocking based on its `stGoInfo` object, it may unblock `g` in the future. Thus, the algorithm immediately returns a false result for this bug-detection attempt at line 7. Otherwise, the algorithm further iterates over all the primitives that `go` is waiting for with the inner loop at lines 10–17. When `go` is blocking at a `select`, the algorithm considers it to be waiting for all channels whose operations belong to the `select`. For all other cases (*e.g.*, waiting to send to a channel, waiting to acquire a lock), the algorithm identifies `go` only waits for one primitive. In an inner-loop iteration, if the target primitive has not been inspected before, the algorithm adds all the goroutines that hold a reference to or have acquired the primitive into Go_{list} with the help of `stPInfo` objects.

Algorithm 1 Blocking Bug Detection

```
Require: goroutine (g) and channel (c) /* g blocks on c */
 1: function (g, c)
 2:     VisitedPrim_set, VisitedGo_set ← {c}, {}
 3:     Go_list ← stPInfo_map[c].getGos()
 4:     while Go_list is not empty do
 5:         go ← Go_list.pop_front()
 6:         if not stGoInfo_map[go].blocking then
 7:             return False, {}
 8:         end if
 9:         VisitedGo_set.insert(go)
10:         for each primitive (p) in stGoInfo_map[go].getPrims() do
11:             if p not in VisitedPrim_set then
12:                 VisitedPrim_set.insert(p)
13:                 for each goroutine (g') in stPInfo_map[p].getGos() do
14:                     Go_list.append(g')
15:                 end for
16:             end if
17:         end for
18:     end while
19:     return True, VisitedGo_set
20: end function
```

The algorithm detects a bug if it reaches line 19. In that case, it returns all the identified blocking goroutines. The sanitizer provides more information to programmers to assist the bug validation and inspection, like where the goroutines are blocking and the goroutines' call stacks.

When to Detect? Algorithm 1 introduces nontrivial overhead due to various checks. To reduce the overhead while reporting bugs in a timely manner, the sanitizer launches the detection in two cases: every second during the execution and when the main goroutine terminates. If GFuzz conducts multiple bug-detection attempts in a single run, it will check whether previously identified blocking goroutines still exist in latter attempts for the purpose of validation.

Working Example. We assume the sanitizer performs Algorithm 1 on the code in Figure 1, when the child goroutine blocks at line 27 and the parent has returned from function `parent()`. We further assume the algorithm takes channel `ch` and the child goroutine as its inputs. The child is added to Go_{list} at line 3 in Algorithm 1, since it is the only goroutine that holds a reference to `ch`. The parent goroutine's reference to `ch` is removed when it returns from function `parent()`. The first iteration of the outer loop inspects the child goroutine and does not append anything to Go_{list}. Then the algorithm leaves the loop and detects the bug at line 19 with $VisitedGo_{set}$ containing only the child goroutine.

7 EVALUATION

We implement GFuzz using Go-1.16. We conduct static analysis using the SSA package [18], perform source-code instrumentation using the AST package [23], and implement the sanitizer by modifying the Go runtime. Our implementation contains 7127 lines of Go code and 1004 lines of Python/Shell scripts. All our experiments are performed on a server machine with Intel(R) Core(TM) i7-8700K CPU and 32GB RAM.

We largely reuse the benchmarks in the GCatch project [43] to conduct our experiments. Among the ten applications where GCatch finds previously unknown channel-related bugs, we evaluate GFuzz on seven of them. For the other three applications, we do not evaluate the Go project since our modification to the Go

	Benchmark Info.			Detected New Bugs							
App	**Star**	**LoC**	**Test**	**chan$_b$**	**select$_b$**	**range$_b$**	**NBK**	**Total**	**GFuzz$_3$**	**GCatch**	**Overhead$_s$**
Kubernetes	74K	3453K	3176	28	4	9	2	43	18	3	36.75%
Docker	60K	1105K	1227	17	2	-	-	19	5	4	44.53%
Prometheus	35K	1186K	570	14	-	1	3	18	8	-	18.08%
etcd	35K	181K	452	7	12	-	1	20	7	5	14.43%
Go-Ethereum	28K	368K	1622	11	43	6	2	62	40	5	75.18%
TiDB	27K	476K	264	-	-	-	-	-	-	-	17.65%
gRPC	13K	117K	888	15	-	1	6	22	7	8	20.00%
Total	272K	6887K	8199	92	61	17	14	184	85	25	/

Table 2: Benchmarks and Evaluation Results. ***LoC means lines of source code; test denotes number of unit tests used in our experiments. In the "Detected New Bugs" columns, subscript b denotes blocking bugs; NBK represents non-blocking bugs; "-" means zero bugs; and GFuzz$_3$ represents bugs detected in the first three fuzzing hours. The "Overhead$_s$" column shows the overhead of the sanitizer and "/" means not applicable.***

runtime brings in conflicting dependencies when running GFuzz on Go. We fail to build some dependent C libraries of CockroachDB and thus we do not evaluate GFuzz on it. We also intentionally skip bblot as it is very small and does not contain many synchronization operations.

Table 2 shows the information of the seven evaluated applications. The applications cover different types of functionalities (*e.g.*, container systems, databases, RPC libraries) and represent the typical usage of Go when implementing server-side software. They are popular, and most of them are ranked within the top 20 based on the number of GitHub stars they have received. All evaluated applications have more than five years' development history and are still under active development. All of them have a large program size, with three containing more than one million lines of source code. These applications can help validate the efficacy of GFuzz for testing large, real Go software.

We evaluate GFuzz from four aspects. ① *Effectiveness*: how many new bugs does GFuzz detect? (Section 7.1) ② *Advancement*: does GFuzz detect more bugs than the state-of-the-art Go concurrency bug detector GCatch? (Section 7.2) ③ *Necessity*: how does each component of GFuzz contribute to bug detection? (Section 7.3) ④ *Performance*: what is the runtime overhead incurred by GFuzz? (Section 7.4)

7.1 Effectiveness of Bug Detection

Methodology. To access GFuzz's effectiveness, we apply it to multiple recent application versions and count how many bugs and false positives are reported. Our instrumentation makes some unit tests fail to be compiled. Thus, we only use those tests that can be compiled after the instrumentation as seeds to run GFuzz. Table 2 shows the detailed number of used tests for each application. We initially configure GFuzz to wait for 500ms[3] to prioritize a message (*e.g.*, T in Figure 3). If GFuzz fails to wait for any message in one run, it increases T by three seconds and adds the order back to the order queue. The Go testing framework kills a unit test if it cannot finish in 30 seconds. By default, we use five workers to run GFuzz.

[3]We have tried 250ms, 500ms, and 1000ms on gRPC, and 500ms returns the best results.

```
1  func parent() {
2      stopChan := make(chan struct{})
3      ca := &cloudAllocator{
4          nodeUpdateChannel: make(chan string, 1),
5      }
6      go ca.worker(stopChan)
7      ... // neither nodeChannel nor stopChan is closed
8  }
9  func (ca *cloudAllocator) worker(stopChan <- chan struct{}) {
10     for {
11         select {
12         case workItem, ok := <-ca.nodeUpdateChannel:
13             if !ok {
14                 Log("Unexpectedly_Closed")
15                 return
16             }
17             ... // process node updates
18         case <-stopChan:
19             return
20         }
21     }
22 }
```

Figure 5: A Bug Blocking at a `select` Statement. ***The code has been simplified for illustration purposes.***

Those workers execute unit tests concurrently, but their accesses to the order queue are sequentialized.

Effectiveness Results. As shown in Table 2, GFuzz detects 184 previously unknown bugs, including 170 blocking bugs and 14 non-blocking bugs, and reports 12 false positives. We have filed bug reports for all the detected bugs. As of this writing, programmers have confirmed 124 bugs as real bugs and fixed 67 of them based on our reporting. The large number of detected bugs confirms the effectiveness of GFuzz.

All blocking bugs are identified by our sanitizer and are missed by the Go runtime. All of them are caused by one single blocking goroutine and do not involve circular wait among multiple goroutines. Among them, 92 bugs render the buggy goroutine blocking at a channel operation (sending or receiving). An example of such a bug is shown in Figure 1.

61 blocking bugs make the buggy goroutine block at a `select` and wait for multiple channel operations. For example, the parent goroutine in Figure 5 creates a child goroutine at line 6, which keeps processing updates sent through channel `ca.nodeUpdateChannel` with a `select` (lines 11–20) in a loop until the moment when either

```
1  func parent(queueLength int) *Broadcaster {
2      m := &Broadcaster{
3          incoming: make(chan Event, queueLength),
4      }
5      go m.loop()
6      return m
7  }
8  func (m *Broadcaster) loop() {
9      for event := range m.incoming { //blocking
10         m.distribute(event)
11     }
12 }
13
14 func (m *Broadcaster) Shutdown() {//forget to be called
15     close(m.incoming)
16 }
```

Figure 6: A Bug Blocking at a `range` Statement. ***The code has been simplified for illustration purposes.***

channel `ca.nodeUpdateChannel` or channel `stopChan` is closed. However, the parent does not close either of the two channels, and thus the child goroutine blocks at the `select` and waits for messages from the two channels endlessly.

17 bugs leave a goroutine blocking when using a `range` to pull messages from a channel. The `range` keyword can drain a channel in a loop and keeps iterating the loop until the channel is closed. For example, programmers forget to call function `Shutdown()` at line 14 in Figure 6, which causes the child goroutine created at line 6 to be blocked at the range statement at line 9 forever.

Moreover, GFuzz detects 14 non-blocking bugs. All of them are captured by the Go runtime. Although those bugs are not reported by our sanitizer, reordering concurrent messages is necessary to trigger a code path or a goroutine interleaving for them. One of the bugs is caused by sending a message to a closed channel, two are caused by using an out-of-bound index to access a slice or an array, nine are due to dereferencing a `nil` object, and the root cause of the remaining two bugs is accessing a `map` without any synchronization.

All the reported false positives are caused by errors of GFuzz's static analysis. GFuzz misses some code sites where a goroutine gains a reference to a channel and fails to instrument `GainChRef()` calls there. Thus, for some goroutines, GFuzz does not know they have gained some channel references until they conduct operations on the channels. If the testing framework terminates a testing run in such a time window, using the incomplete relations between goroutines and channel references, GFuzz may mistakenly determine that no active goroutines can unblock some blocking goroutines and report false alarms. All false positives occur for this reason.

7.2 Comparison with GCatch

Methodology. We compare GFuzz with GCatch [43], the most recently developed Go concurrency bug detector, to understand whether GFuzz advances the state of the art. GCatch models channel operations using a new constraint system and extracts constraints through static program analysis. It leverages Z3 [8] to solve the constraints to look for goroutine interleavings that lead one or more goroutines to block endlessly and thus detects blocking bugs. We apply GFuzz and GCatch to the same version of each evaluated application. We compare the detected bugs of GCatch and the bugs reported by GFuzz in the first three hours to understand the detectors' pros and cons.

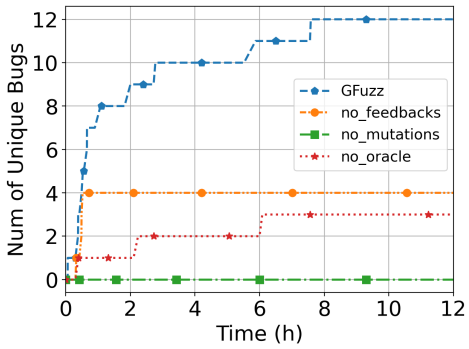

Figure 7: Contributions of GFuzz Components.

Comparison Results. In total, GFuzz finds 85 bugs in three hours and GCatch detects 25 bugs in the same application versions.

Among the 85 bugs found by GFuzz, GCatch also pinpoints five of them and fails to report the others for four reasons. First, GCatch does not detect non-blocking bugs. Thus, it misses four bugs. Second, when GCatch conducts inter-procedural analysis, if a call site may have more than one callee, GCatch gives up the analysis. Thus, GCatch misses some synchronization operations and fails to detect 57 bugs. Note that GCatch takes this design to retain its precision, and if it continues to analyze call sites with more than one possible callee, it reports many false positives. Third, GCatch fails to pinpoint 17 bugs because it does not have some necessary dynamic information, such as channel buffer size and which channel a pointer points to. Fourth, GCatch fails to analyze loop iteration numbers for the remaining two bugs. Compared with GCatch, GFuzz overcomes several fundamental limitations of static analysis (*e.g.*, alias analysis, the computation of loop iteration numbers). It significantly improves the state of the art of Go concurrency bug detection.

For the 25 bugs detected by GCatch, GFuzz does not find 20 of them in the three allotted hours due to four reasons. First, GFuzz could have detected six more bugs, but doing so would require a longer execution time. Second, for four bugs, reordering concurrent messages does not help expose them. For example, one bug identified by GCatch can only be triggered when a function returns a particular value. However, changing message orders cannot help the function return the required value. Third, for eight bugs, there are no unit tests available to exercise the buggy code or the tests require particular execution environments that are unavailable to us. Fourth, the remaining two bugs are missed because GFuzz cannot handle some control labels during source-to-source transformation.

7.3 Necessity of GFuzz Components

Methodology. To understand the contribution made by each component of GFuzz, specifically, order mutation, order prioritization, and bug detection, we launch a dynamic evaluation to detect bugs from gRPC with each component disabled. We choose gRPC version 9280052 (committed on 7 February, 2021), the most recent gRPC

version used in our experiments, to conduct this evaluation. We use five workers for all settings.

Experimental Results. Figure 7 shows the number of unique bugs detected by each setting of GFuzz over the first 12 hours. In total, 14 unique bugs are detected across the four settings. Among them, four bugs are fixed by programmers themselves before we file our reports, and the remaining ten bugs are included in the "Total" column of Table 2.

First, the full-featured GFuzz finds the highest number of unique bugs for a total of 12. Nine of these bugs are channel-related blocking bugs (reported by our sanitizer). The remaining three are caused by dereferencing a nil pointer (pinpointed by the Go runtime).

Second, without the bug sanitizer, the Go runtime cannot report any blocking bugs even if the bugs are triggered. However, the Go runtime captures three non-blocking bugs, including two nil-pointer dereferences and one caused by concurrent accesses on a map. Since the testing process contains some randomness, the detected non-blocking bugs are not exactly the same as those detected in the full-featured run. The two groups of bugs have two nil-pointer dereferences in common. The full-featured GFuzz pinpoints one more nil-pointer dereference, but it misses the bug due to unsynchronized accesses on a map.

Third, without any order mutation, GFuzz cannot detect any concurrency bugs. This dynamic execution further confirms the benefits of message reordering.

Fourth, runtime feedback can help find more bugs than pure random testing. Without feedback, GFuzz can find only four bugs, including one blocking bug and three non-blocking bugs. Three of those bugs are also detected by the full-featured GFuzz. One bug due to concurrent accesses on a map is missed in the full-featured run. We also observe that without feedback, GFuzz cannot find any bugs after one hour. Since the mutation space is huge, it is inefficient to blindly explore the space.

7.4 Performance

Overhead of GFuzz. We run GFuzz on each application for 12 hours to measure execution speed. We launch five workers for GFuzz. For the purposes of comparison, we run the same set of unit tests using the testing framework with parallel level five.

GFuzz can execute 0.62 unit tests in one second for the applications and causes 3.0X overhead. GFuzz causes runtime overhead for two reasons. First, GFuzz adds extra waits when prioritizing a particular concurrent message. Second, GFuzz collects various types of dynamic information for detecting bugs and prioritizing interesting message orders.

Overhead of the Sanitizer. We particularly measure the runtime overhead of the sanitizer to understand the feasibility of adopting it in other scenarios. To achieve this goal, we disable the instrumented code that reorders concurrent messages and that collects information to guide the fuzzing method. Some (but not all) tests in Prometheus and gRPC have blocking detection functionality, which will wait for several extra seconds if goroutines not in their whitelists are still live when the main goroutine finishes. We disable such functionality in our measurement. We run all unit tests *10* times with and without the sanitizer and then compute the overhead using the average execution time.

As shown in Table 2, the sanitizer incurs less than 20% overhead for two applications (Prometheus and etcd) and less than 50% overhead for another four applications (Kubernetes, Docker, TiDB, and gRPC). The sanitizer causes 75.2% overhead for Go-Ethereum, which is the largest overhead incurred among all the applications. Overall, the sanitizer causes overhead less than or comparable with widely-used sanitizers, such as AddressSanitizer [59] and ThreadSanitizer [60].

8 DISCUSSION AND FUTURE WORK

Limitations and Future Work. GFuzz alters program execution states only by mutating the processing order of concurrent messages. However, there are many other mechanisms to change Go programs' executions, such as mutating program inputs, changing the order of shared-memory accesses, and modifying the interleaving of synchronization operations. We will explore how these mechanisms impact the exposure of Go concurrency bugs and enhance GFuzz accordingly.

GFuzz only considers messages waited for by the same `select` as concurrent, and thus it misses many other possible concurrent messages. We believe reordering these missed messages can reveal more channel-related programming mistakes and detect more bugs. We will explore advanced analysis techniques to detect more concurrent messages in the future.

GFuzz does not modify program inputs, and thus its effectiveness depends on the code coverage of the program inputs used to launch GFuzz. We use existing unit tests (not program inputs) to conduct our experiments. We do observe some bugs that are detected by the static detector GCatch but are missed by GFuzz, because there is no unit test to exercise the buggy code. We consider this a fundamental limitation of many dynamic techniques. In the future, we will look for real-world workloads of the benchmark applications and test how GFuzz works with those workloads.

GFuzz triggers high overhead, which can cause both false positives and false negatives. First, the overhead can lead a unit test not to finish in 30 seconds and to be killed by the testing framework. As discussed in Section 7.1, GFuzz may not record the precise goroutine-channel relations under this scenario, and thus it may report false positives. Second, some goroutines may be significantly slowed down by GFuzz, and some bugs may not be triggered anymore. Thus, GFuzz misses those bugs. In the future, we will explore how to speed up GFuzz and improve both the effectiveness and the efficiency of GFuzz.

Generalization to Other Programming Languages. Since many new programming languages adopt message passing to reduce concurrency bugs, extending GFuzz to these languages after some adjustments appears promising. For example, both Rust and Kotlin support `select` to enable one thread to wait for multiple channel operations. Thus, the way GFuzz identifies concurrent messages can also be used to pinpoint concurrent messages in Rust and Kotlin programs. As another example, GFuzz's blocking-bug detection algorithm can likewise be used to detect blocking bugs in Rust and Kotlin programs after two modifications. First, a channel in a Rust program by default has an unlimited buffer size, and thus

the algorithm should be modified to not consider that a sending operation can block a thread. Second, Kotlin organizes threads in a hierarchical way, and when a parent thread terminates, all child threads will also be stopped. Thus, the algorithm should be enhanced to consider that a parent thread can potentially unblock all its child threads.

9 RELATED WORK

Dynamic Concurrency Bug Detection. Go provides a built-in deadlock detector, which is implemented in the goroutine scheduler and is always enabled [17]. The detector detects some channel-related blocking bugs, but it misses most of them because it reports bugs only when *all* (not some) goroutines are blocked at synchronization operations. For example, none of the previously unknown blocking bugs found by GFuzz (including those presented in Figure 1, Figure 5, and Figure 6) can be identified by the deadlock detector. Industry practitioners build techniques that report blocking bugs when there are goroutines finishing later than the main goroutine [6, 67]. Since Go is mainly used to implement server-side software, a Go program can execute for a long time. Bug detection is thus significantly delayed by the techniques. Moreover, all the existing techniques cannot increase the chance of triggering a concurrency bug.

GFuzz is similar to previous techniques [1, 11, 39, 48, 53, 58] built for other programming languages in a way that they all force threads (or goroutines) to interact in particular ways and report bugs that are validated at runtime. However, GFuzz focuses on concurrent messages, not shared-memory usages. Since messages are usually processed by many different channels and there is no single memory location to identify correlated messages, analyzing messages is more challenging than inspecting shared-memory accesses. Moreover, GFuzz leverages the fuzzing method to identify and prioritize message orders that are close to exposing bugs.

Researchers have built techniques to identify concurrency bugs in distributed systems. Some of these techniques use model checking to systematically examine all possible message orders for possible bugs [38, 46, 72]. Since a real Go software system has a huge number of possible message orders, we think effectively searching the message-order space of the system that GFuzz offers is more efficiently to detect channel-related bugs in the system than exhaustively inspecting the whole order space. After modeling concurrency mechanisms in distributed systems using happens-before rules, DCatch can effectively pinpoint concurrency bugs using runtime tracing and trace analysis [41]. However, DCatch focuses on bugs caused by concurrent conflicting memory accesses on one node and its algorithm cannot be used to detect channel-related bugs. Morpheus samples and enforces partial orders of conflicting messages (*i.e.*, concurrent messages sent to the same process) to detect bugs in distributed systems implemented in Erlang [74]. Similar to Morpheus, GFuzz also mutates orders of concurrent messages to expose bugs. Unlike Morpheus, GFuzz leverages channel states (*e.g.*, number of channel elements, closed or not) to more efficiently explore the order space.

There are dynamic detectors that detect deadlocks in MPI programs [12, 24, 25, 62, 68], where MPI (message passing interface) is a library that helps create parallel programs in C or Fortran77. However, MPI channels are different from Go channels in their design model, and many important Go channel operations (*e.g.*, close, `select`) do not exist in MPI. Therefore, the detectors built for MPI programs cannot effectively identify channel-related bugs in Go.

Static Concurrency Bug Detection for Go. Practitioners and researchers have built many techniques to statically detect concurrency bugs in Go programs. Staticcheck [27] and the vet tool [15] are two suites of static Go bug detectors, each containing four detectors for identifying concurrency bugs. However, these tools target very specific buggy code patterns (*e.g.*, an unlocking operation right after a locking operation on the same `mutex`), and thus they miss most Go concurrency bugs.

Researchers have built several model-checking-based techniques that extract a Go program's execution model by inspecting synchronization operations and detect liveness issues (*i.e.*, blocking bugs) and channel safety issues (*i.e.*, non-blocking bugs) [13, 36, 37, 51, 57]. However, these techniques have to analyze each input program and all its primitives as a whole, and thus it is difficult for them to scale to large, real Go software.

GCatch is the most recent Go concurrency-bug detector [43]. GCatch separates synchronization primitives into small groups, and only examines each group within a small code scope to scale to large programs. After modeling channel operations in a novel constraint system, GCatch applies a constraint solver to detect bugs. However, GCatch can only detect channel-related blocking bugs. It is not easy to extend GCatch's constraint system to cover more primitives and non-blocking bugs, since we need to use constraints to precisely model those primitives' behaviors and non-blocking bugs' triggering conditions. In addition, to report fewer false positives, GCatch gives up its static analysis when one call site has multiple possible callees, which causes GCatch to miss many bugs in our experiments.

Overall, there are several fundamental difficulties in statically analyzing real-world Go software, including computing precise alias information, constructing accurate call graphs when interfaces are involved, and calculating loop iteration numbers. Thus, we design GFuzz using dynamic analysis to work around these challenges.

Grey-box Fuzzing. Fuzzing is a general technique used to stress the tested program by randomly generating a lot of inputs [47]. Many advanced fuzzing techniques have identified hundreds of thousands of previously unknown bugs and vulnerabilities in real-world software systems [5, 44, 55, 64, 73, 75, 76]. There are also fuzzers built for Go programs [16, 26, 69]. However, most of them focus on bugs that can be triggered in single-thread mode and are not designed for concurrency bugs.

Several fuzzing techniques have been designed to effectively detect concurrency bugs by prioritizing seed inputs that can trigger more concurrent code or particular interleavings [4, 32, 40]. Although useful, these techniques target concurrency bugs due to misuse of shared memory and are thus unlikely to detect channel-related bugs.

10 CONCLUSION

This paper presents a new detection technique GFuzz that provides push-button, accurate bug detection for channel-related Go

concurrency bugs by proactively mutating the processing order of concurrent messages. GFuzz identifies concurrent messages using a straightforward approach, safely enforces required processing orders of concurrent messages, selectively prioritizes promising message orders, and detects resulting concurrency bugs. In our experiments, GFuzz successfully found 184 previously unknown bugs from six popular Go applications, outperforming the state-of-the-art Go concurrency bug detector. Future research can further explore how to leverage other execution mutation mechanisms to enhance GFuzz and identify more concurrent messages.

ACKNOWLEDGEMENT

We thank Yatin Manerkar, our shepherd, and the anonymous reviewers for their insightful feedback and comments. We thank Boqin Qin for helping file bug reports. This work is supported in part by NSF grant CNS-1955965.

REFERENCES

[1] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. Pittsburgh, Pennsylvania, USA.

[2] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-Scale Applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. Zurich, Switzerland.

[3] Yan Cai, Shangru Wu, and W. K. Chan. 2014. 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.

[4] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *Proceedings of 29th USENIX Security Symposium (USENIX Security '20)*. Virtual Event, USA.

[5] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposiums on Security and Privacy (Oakland '18)*. San Francisco, CA, USA.

[6] CockroachDB. 2021. leaktest. https://github.com/cockroachdb/cockroach/tree/master/pkg/util/leaktest.

[7] Randy Coulman. 2013. Debugging Race Conditions and Deadlocks. https://randycoulman.com/blog/2013/03/05/debugging-race-conditions-and-deadlocks/.

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Berlin, Heidelberg.

[9] Nicolas Dilley and Julien Lange. 2019. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. Hangzhou, China.

[10] Docker. 2022. Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/.

[11] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. Vancouver, BC, Canada.

[12] Vojtundefinedch Forejt, Saurabh Joshi, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2017. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. *ACM Transactions on Programming Languages and Systems* (2017).

[13] Julia Gabet and Nobuko Yoshida. 2020. 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.

[14] Andrew Gerrand. 2010. The Go Blog: Share Memory By Communicating. https://blog.golang.org/share-memory-by-communicating.

[15] Google. 2021. Command vet. https://golang.org/cmd/vet/.

[16] Google. 2021. Fuzz testing for go. https://github.com/google/gofuzz.

[17] Google. 2021. Package Deadlock. https://godoc.org/github.com/sasha-s/go-deadlock.

[18] Google. 2021. Package SSA. https://godoc.org/golang.org/x/tools/go/ssa.

[19] Google. 2022. A high performance, open-source universal RPC framework. https://github.com/grpc/grpc-go.

[20] Google. 2022. Data Race Detector. https://golang.org/doc/articles/race_detector.html.

[21] Google. 2022. Effective Go: Concurrency. https://golang.org/doc/effective_go.html#concurrency.

[22] Google. 2022. Honggfuzz. https://google.github.io/honggfuzz/.

[23] Google. 2022. Package AST. https://golang.org/pkg/go/ast/.

[24] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. 2009. A Graph Based Approach for MPI Deadlock Detection. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. Yorktown Heights, NY, USA.

[25] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. 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.

[26] Katie Hockman. 2021. Design Draft: First Class Fuzzing. https://golang.org/s/draft-fuzzing-design.

[27] Dominik Honnef. 2022. Staticcheck — a collection of static analysis tools for working with Go code. https://github.com/dominikh/go-tools.

[28] Jeff Huang. 2015. 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)*. Portland, OR, USA.

[29] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. 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)*. Edinburgh, United Kingdom.

[30] Jeff Huang and Arun K. Rajagopalan. 2016. 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)*. Amsterdam, Netherlands.

[31] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. 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.

[32] Dae R. Jeong, Kyungtae Kim, Basavesh Ammanaghatta Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Proceedings of the 40th IEEE Symposiums on Security and Privacy (Oakland '19)*. San Francisco, CA, USA.

[33] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. 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.

[34] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. 2016. Sound static deadlock analysis for C/Pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. Singapore, Singapore.

[35] Kubernetes. 2022. Production-Grade Container Orchestration. https://kubernetes.io/.

[36] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2017. 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)*. Paris, France.

[37] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden.

[38] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI '14)*. Broomfield, CO, USA.

[39] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. 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)*. Huntsville, Ontario, Canada.

[40] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. San Juan, PR, USA.

[41] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Xi'an, China.

[42] Tapir Liu. 2022. Some Panic/Recover Use Cases. https://go101.org/article/panic-and-recover-use-cases.html.

[43] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems.

In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Virtual Event, USA.

[44] LLVM. 2022. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[45] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. 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)*. San Jose, California, USA.

[46] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Dresden, Germany.

[47] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44.

[48] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. San Diego, California, USA.

[49] Mayur Naik and Alex Aiken. 2007. 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)*. Nice, France.

[50] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. Ottawa, Ontario, Canada.

[51] Nicholas Ng and Nobuko Yoshida. 2016. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. Barcelona, Spain.

[52] Stack Overflow. 2020. Stack Overflow Developer Survey 2020. https://insights.stackoverflow.com/survey/2020.

[53] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. 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)*. Washington, DC, USA.

[54] Dawson R. Engler and Ken Ashcraft. 2003. 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.

[55] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS '17)*. San Diego, CA, USA.

[56] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. 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.

[57] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. Phoenix, AZ, USA.

[58] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Tucson, AZ, USA.

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. Boston, MA.

[60] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71.

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

[62] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. 2012. 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.

[63] Alan Shreve. 2014. Principles of designing Go APIs with channels. https://inconshreveable.com/07-08-2014/principles-of-designing-go-apis-with-channels/.

[64] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS '16)*. San Diego, CA, USA.

[65] Ashish Tiwari. 2020. Golang fatal error: concurrent map writes. https://ashish.one/blogs/fatal-error-concurrent-map-writes/.

[66] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. 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)*. Providence, RI, USA.

[67] Uber. 2021. goleak. https://github.com/uber-go/goleak.

[68] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. 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.

[69] Dmitry Vyukov. 2021. go-fuzz: randomized testing for Go. https://github.com/dvyukov/go-fuzz.

[70] Jacob Walker. 2018. Goroutine Leaks - The Forgotten Sender. https://www.ardanlabs.com/blog/2018/11/goroutine-leaks-the-forgotten-sender.html.

[71] Wikipedia. 2022. Go (programming language). https://en.wikipedia.org/wiki/Go_(programming_language).

[72] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*. Boston, Massachusetts, USA.

[73] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, Xiaofeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *Proceedings of the 40th IEEE Symposiums on Security and Privacy (Oakland '19)*. San Francisco, CA, USA.

[74] Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Lausanne, Switzerland.

[75] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security '18)*. Berkeley, CA, USA.

[76] Michal Zalewski. 2020. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.