Who Goes First? Detecting Go Concurrency Bugs via Message Reordering

Ziheng Liu*, **Shihao Xia***, Yu Liang, Linhai Song and Hong Hu

Pennsylvania State University

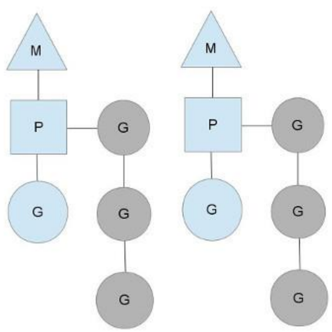
* Ziheng Liu and Shihao Xia are co-first authors

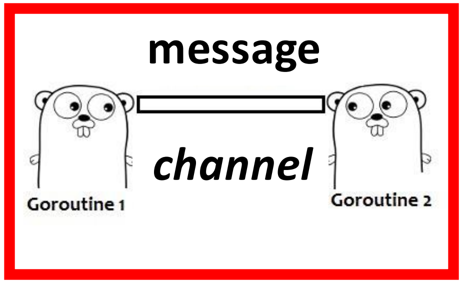
PennState College of Information Sciences and Technology

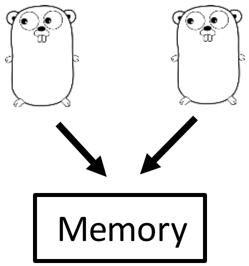
Go Programming Language

- A young but widely-used programming lang.

- Designed for efficient and *reliable* concurrency
  - Provide lightweight threads, called goroutines
  - Support both *message passing* and shared memory







Concurrency Bugs in Go

- Many concurrency bugs are in Go programs

docker

Remove deadlock in ContainerWait #33293

LCOW: Graphdriver fix deadlock in hotRemoveVHDs #36114

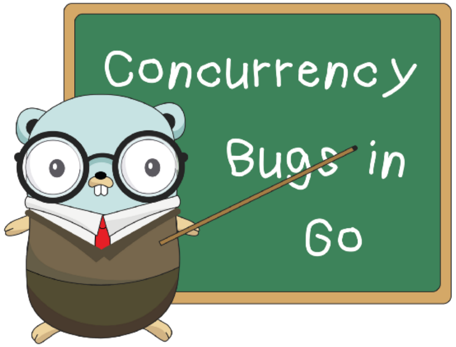
...xy #7559

on 13 Aug 2014

Prevent a goroutine leak when healthcheck gets stopped #33781

Merged aaronlehmann merged 1 commit into moby:master from mlaventure:fix-healchcheck-goroutine-leak on 26 Jun 2017

- Channel may be more error-prone than mutex
  - 36% more blocking bugs are due to channels [1]



[1] Tengfei Tu, Xiaoyu Liu, Linhai Song and Yiying Zhang. “Understanding Real-World Concurrency Bugs in Go.” In **ASPLOS’19**.

A Concurrency Bug in Docker

Parent Goroutine

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

A Concurrency Bug in Docker

Parent Goroutine

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

after 1 second

a msg from the child

Child Goroutine

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

send a message

A Concurrency Bug in Docker

Parent Goroutine

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

Child Goroutine

unbuffered

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

blocking endless!!

Limitations of Existing Techniques

Parent Goroutine

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e :
    Log("E
  }
  return
}
```

Child Goroutine

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, err
}
```

cannot resolve indirect function calls

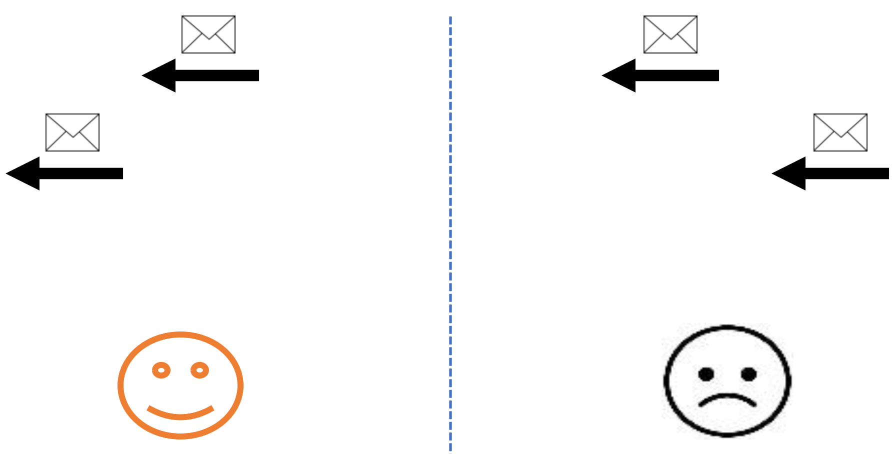
don't increase the chance of exposing concurrency bugs

do not analyze channel operations

Intuitions

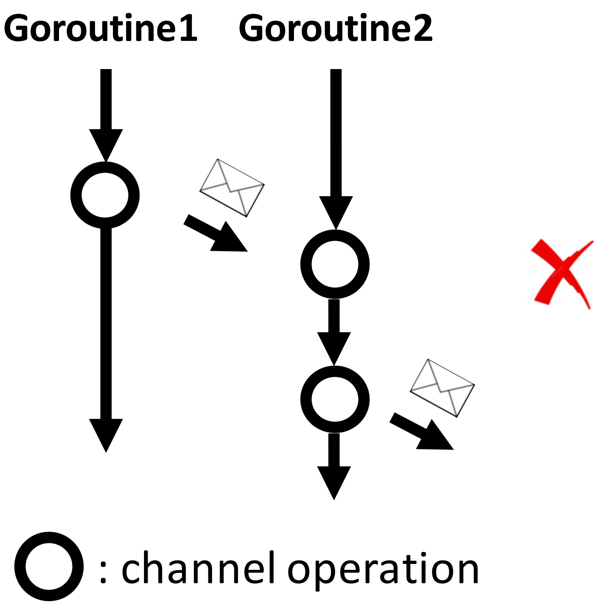
- Building a dynamic technique
  - To avoid limitations of static analysis
- Focusing on concurrent messages
  - Their processing order is non-deterministic
    - Some orders may not be carefully implemented
  - Mutating their processing order to detect bugs

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```



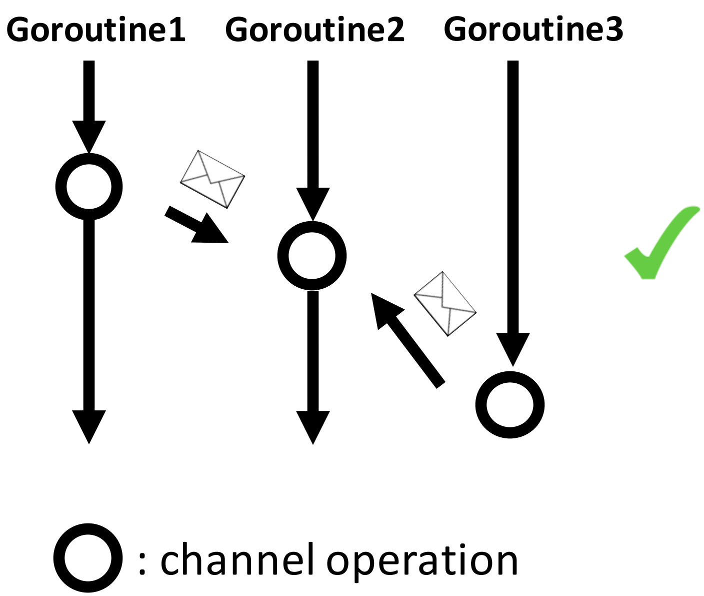
Challenges

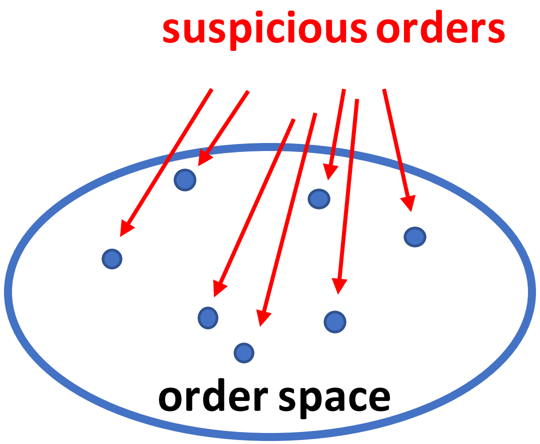
- How to identify concurrent messages?



Challenges

- How to identify concurrent messages?
- How to identify suspicious message orders?
- How to capture triggered channel-related bugs?





Contributions

- GFuzz: a dynamic Go concurrency bug detector
  - Use ***select*** to identify concurrent messages
  - Leverage fuzzing to pinpoint suspicious msg orders
  - Propose a novel sanitizer to capture triggered bugs
- Thorough experiments to evaluate GFuzz
  - Detect ***184*** previously unknown bugs
    - Developers have confirmed 124 bugs and fixed 67 bugs
  - Detect significantly more bugs than SOTA

Outline

- Introduction
- Reordering Concurrent Messages
- Favoring Propitious Orders
- Capturing Triggered Concurrency Bugs
- Implementation and Evaluation
- Conclusion

Outline

- Introduction
- Reordering Concurrent Messages
- Favoring Propitious Orders
- Capturing Triggered Concurrency Bugs
- Implementation and Evaluation
- Conclusion

Concurrent Channel Operations (Ops)

- Channel ops have no happens-before relation
  - Their processed messages are concurrent
  - Challenging to find all concurrent channel ops
- GFuzz focuses on ***select*** statements
  - Select allows a goroutine to wait for ***>1*** channel ops
  - Channel ops within the same select are concurrent

```
select {
case <- Fire(1 * T.Second):
   Log("Timeout!")
case e := <- ch:
   …
case e := <- errCh:
   Log("Error!")
}
```

can happen simultaneously

Encode Concurrent Message Orders

- Use selected cases to represent an order
  - Assign each select a unique ID
  - Allocate a local index to each case
  - An order → $[(s_0, c_0, e_0) \ldots (s_n, c_n, e_n)]$

case index

select id

the number of cases

id: 0

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```

chosen twice

⇨ [(0, 3, 1), (0, 3, 1)]

Code Transformation

```
switch FetchOrder(…) {

}
```

```
select {
case <- Fire(1 * T.Second):
   Log("Timeout!")
case e := <- ch:
   …
case e := <- errCh:
   Log("Error!")
}
```

3 cases

Code Transformation

```
select {
case <- Fire(1 * T.Second):
   Log("Timeout!")
case e := <- ch:
   …
case e := <- errCh:
   Log("Error!")
}
```

3 cases

```
switch FetchOrder(…) {
  case 0:

  case 1:

  case 2:

  default:

}
```

3 cases + 1 default

Code Transformation

```
select {
case <- Fire(1 * T.Second):
   Log("Timeout!")
case e := <- ch:
   …
case e := <- errCh:
   Log("Error!")
}
```

⇨

```
switch FetchOrder(…)  {
  case 0:



  case 1:



  case 2:

  default:

}
```

no order is specified

Code Transformation

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```

```
switch FetchOrder(…) {
  case 0:
    select {
    case <- Fire(1 * T.Second):
      Log("Timeout!")
    case <- time.After(T):
      ……
    }
  case 1:

  case 2:

  default:

}
```

Code Transformation

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```

```
switch FetchOrder(…) {
  case 0:
    select {
    case <- Fire(1 * T.Second):
      Log("Timeout!")
    case <- time.After(T):
      ……
    }
  case 1:

  case 2:

  default:

}
```

Code Transformation

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```

⇨

```
switch FetchOrder(…) {
  case 0:
    select {
    case <- Fire(1 * T.Second):
      Log("Timeout!")
    case <- time.After(T):
      ……
    }
  case 1:

  case 2:

  default:

}
```

Code Transformation

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```

```
switch FetchOrder(…) {
case 0:
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case <- time.After(T):
    ……
  }
case 1:

case 2:

default:

}
```

the message arrives within T

the message does not arrive within T

Code Transformation

```
select {
case <- Fire(1 * T.Second):
  Log("Timeout!")
case e := <- ch:
  …
case e := <- errCh:
  Log("Error!")
}
```

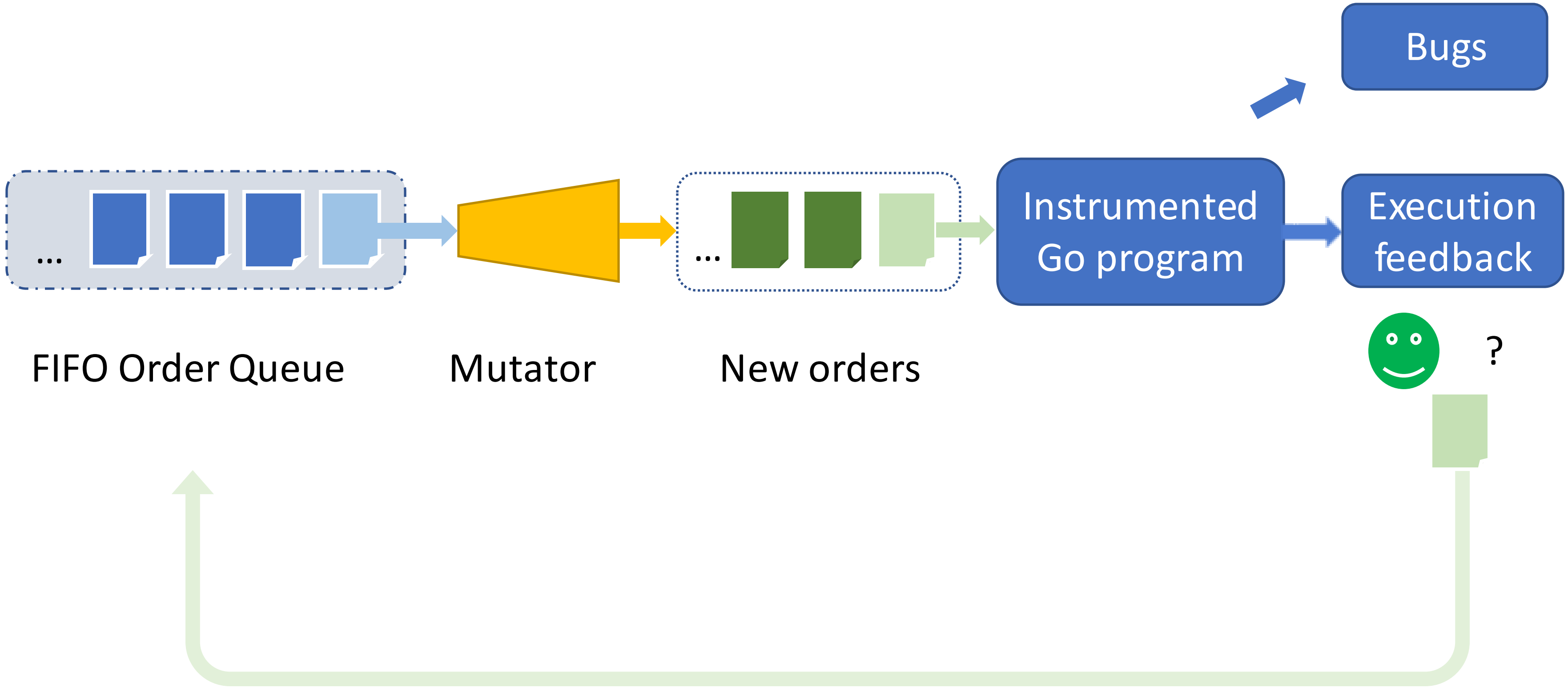
```
switch FetchOrder(…) {
  case 0:
    select {
    case <- Fire(1 * T.Second):
      Log("Timeout!")
    case <- time.After(T):
      ……
    }
  case 1:
    select {
    case e := <- ch:
      ……
    case <- time.After(T):
      ……
    }
  case 2:
    ……
  default:
    ……
}
```

Outline

- Introduction
- Reordering Concurrent Messages
- Favoring Propitious Orders
- Capturing Triggered Concurrency Bugs
- Implementation and Evaluation
- Conclusion

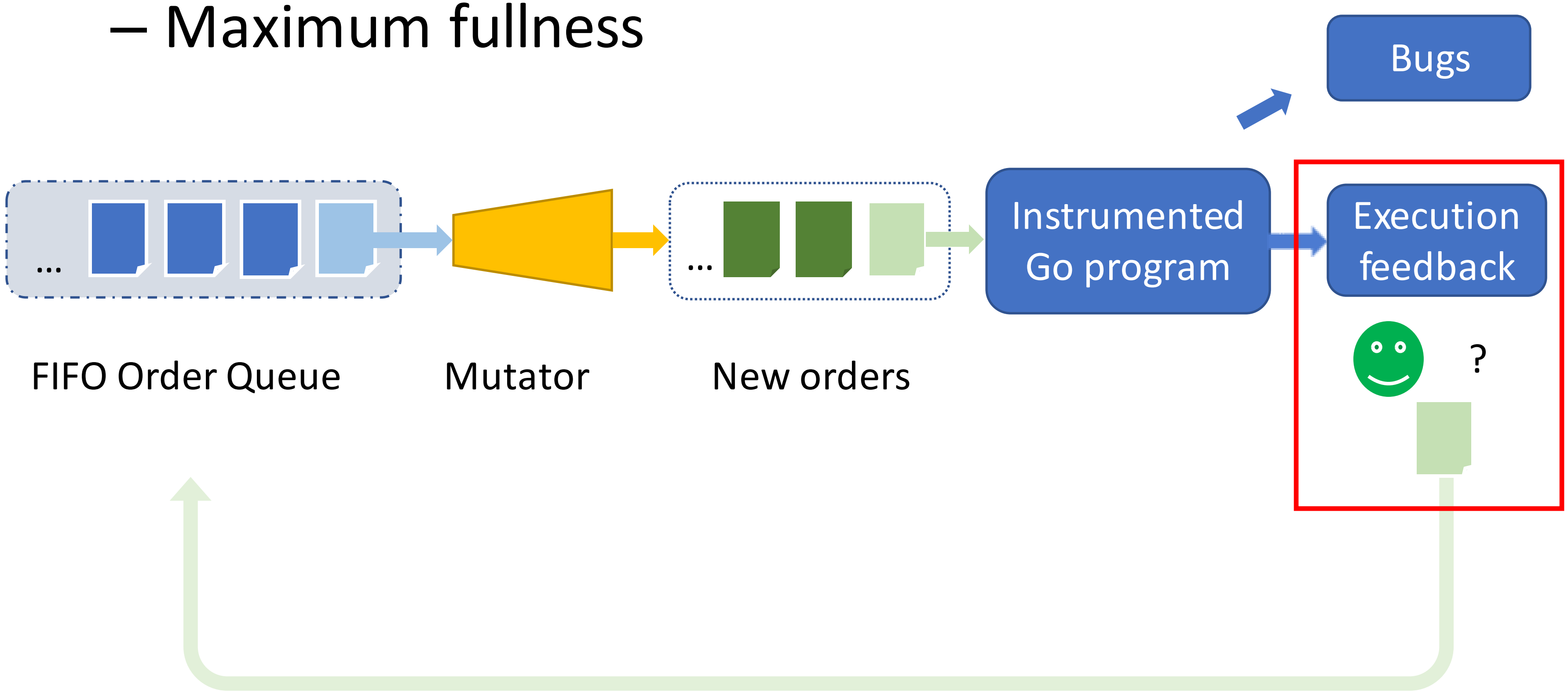
Fuzzing Message Orders

- Continuously mutating exercised orders
- Enforcing new orders to detect bugs



Interesting Orders

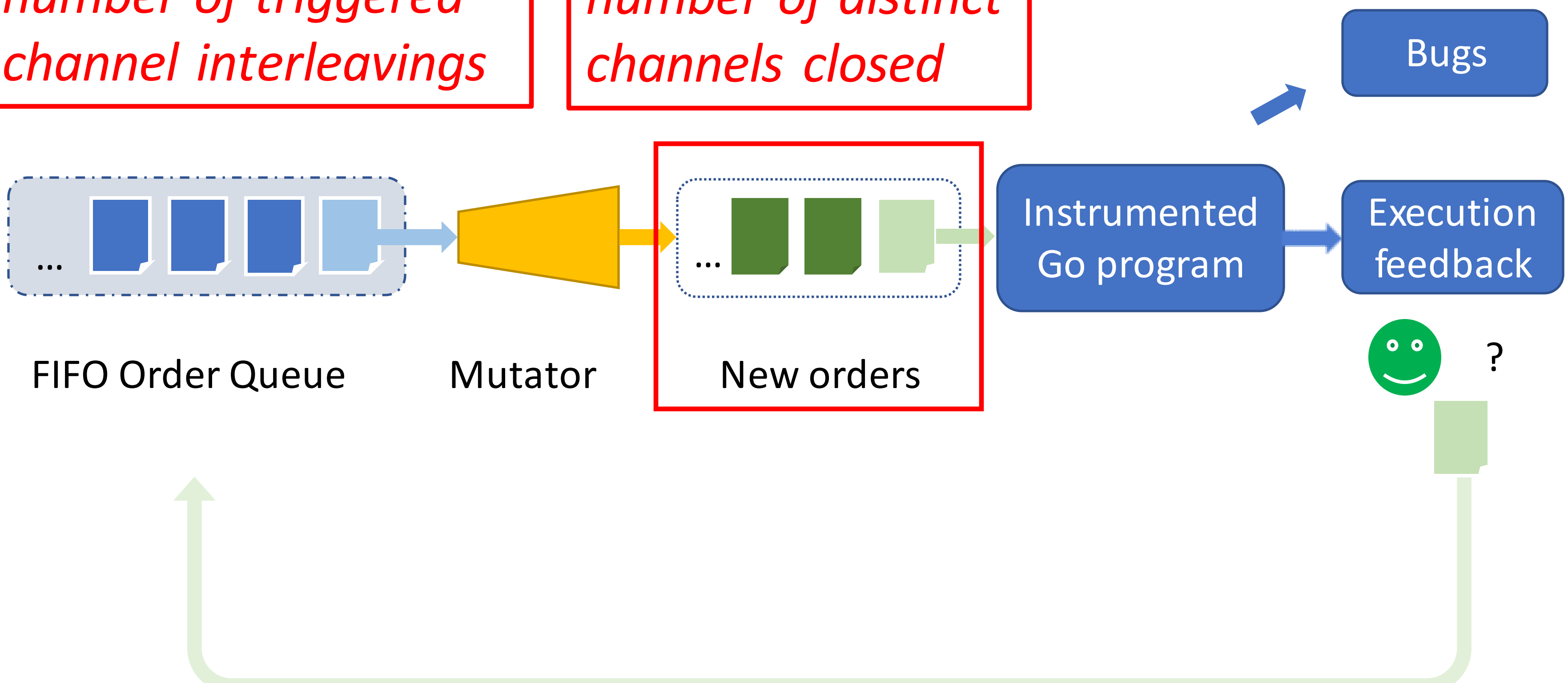
- Triggering a new interleaving of channel ops
- Reaching a new channel state
  - New channel creation, new channel closing
  - Maximum fullness



Prioritizing Valuable Orders

$$score = \sum \log_2 CountChOpPair + 10 * \#CreateCh + 10 * \#CloseCh + 10 * \sum MaxChBufFull$$

- $CountChOpPair$: *number of triggered channel interleavings*
- $\#CreateCh$: number of distinct channel created
- $\#CloseCh$: *number of distinct channels closed*
- $MaxChBufFull$: sum of max channel fullness



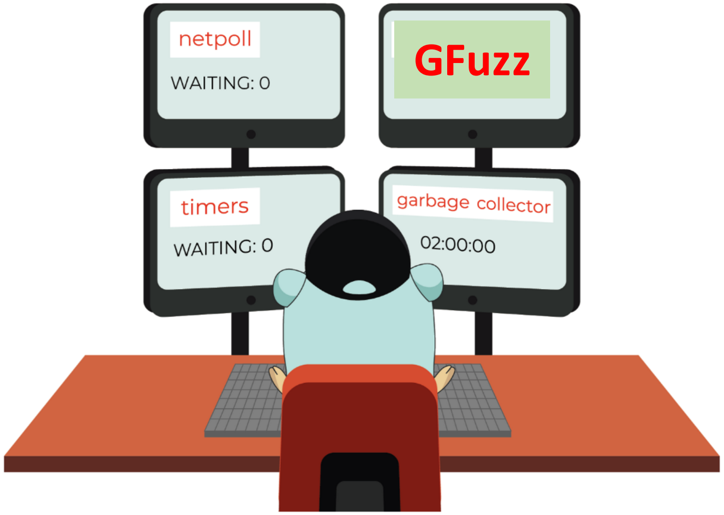
Outline

- Introduction
- Reordering Concurrent Messages
- Favoring Propitious Orders
- Capturing Triggered Concurrency Bugs
- Implementation and Evaluation
- Conclusion

Sanitizer Design

- Focus on channel-related blocking bugs
  - Go runtime captures non-blocking bugs
- Hybridize two methods to record dynamic info.
  - Source-to-source instrumentation
  - Modifying the Go runtime

```
 go func() {
+  GainChRef(ch)
+  GainChRef(errCh)
   entries, err := s.fetch()
   if err != nil {
     errCh <- err
   } else {
     ch <- entries
   } …
 }()
```



Blocking Bug Detection

GoInfo

goroutine	channel

ChInfo

channel	goroutine

Blocking Bug Detection

GoInfo

goroutine	channel
parent	ch, errCh

ChInfo

channel	goroutine
ch	parent
errCh	parent

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

Blocking Bug Detection

GoInfo

goroutine	channel
parent	ch, errCh
child	ch, errCh

ChInfo

channel	goroutine
ch	parent child
errCh	parent child

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

Blocking Bug Detection

GoInfo

goroutine	channel
parent	ch, errCh
child	ch, errCh

ChInfo

channel	goroutine
ch	parent child
errCh	parent child

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

Blocking Bug Detection

GoInfo

goroutine	channel
child	ch, errCh

ChInfo

channel	goroutine
ch	child
errCh	child

```
func parent() {
  …
  ch, errCh := dis.Watch()
  select {
  case <- Fire(1 * T.Second):
    Log("Timeout!")
  case e := <- ch:
    …
  case e := <- errCh:
    Log("Error!")
  }
  return
}
```

```
func (s *Discover) Watch() (…) {
  ch := make(chan Entries)
  errCh := make(chan error)
  go func() {
    entries, err := s.fetch()
    if err != nil {
      errCh <- err
    } else {
      ch <- entries
    } …
  }()
  return ch, errCh
}
```

blocking bug!!

Outline

- Introduction
- Reordering Concurrent Messages
- Favoring Propitious Orders
- Capturing Triggered Concurrency Bugs
- Implementation and Evaluation
- Conclusion

Implementation & Evaluation

- Implementing GFuzz using Go-1.16
  - Leveraging the SSA and AST packages
  - Modifying the Go runtime
- Applying to the recent versions of 7 Go apps

App	Star	LoC	Test
Kubernetes	74K	3453K	3176
Docker	60K	1105K	1227
Prometheus	35K	1186K	570
Etcd	35K	181K	452
Go-Ethereum	28K	368K	1622
TiDB	27K	476K	264
gRPC	13K	117K	888

Effectiveness of Bug Detection

- 184 previously unknown bugs

App	$chan_b$	$select_b$	$range_b$	NBK	Total
Kubernetes	28	4	9	2	43
Docker	17	2	-	-	19
Prometheus	14	-	1	3	18
Etcd	7	12	-	1	20
Go-Ethereum	11	43	6	2	62
TiDB	-	-	-	-	-
gRPC	15	-	1	6	22

Effectiveness of Bug Detection

- 184 previously unknown bugs
  - 170 blocking bugs

App	$chan_b$	$select_b$	$range_b$	NBK	Total
Kubernetes	28	4	9	2	43
Docker	17	2	-	-	19
Prometheus	14	-	1	3	18
Etcd	7	12	-	1	20
Go-Ethereum	11	43	6	2	62
TiDB	-	-	-	-	-
gRPC	15	-	1	6	22

Effectiveness of Bug Detection

- 184 previously unknown bugs
  - 170 blocking bugs

wait for one channel operation

App	$chan_b$	$select_b$	$range_b$	NBK	Total
Kubernetes	28	4	9	2	43
Docker	17	2	-	-	19
Prometheus	14	-	1	3	18
Etcd	7	12	-	1	20
Go-Ethereum	11	43	6	2	62
TiDB	-	-	-	-	-
gRPC	15	-	1	6	22

Effectiveness of Bug Detection

- 184 previously unknown bugs
  – 170 blocking bugs

wait for multiple channel operation

App	$chan_b$	$select_b$	$range_b$	NBK	Total
Kubernetes	28	4	9	2	43
Docker	17	2	-	-	19
Prometheus	14	-	1	3	18
Etcd	7	12	-	1	20
Go-Ethereum	11	43	6	2	62
TiDB	-	-	-	-	-
gRPC	15	-	1	6	22

Effectiveness of Bug Detection

- 184 previously unknown bugs
  - 170 blocking bugs

drain a channel in a loop

App	$chan_b$	$select_b$	$range_b$	NBK	Total
Kubernetes	28	4	9	2	43
Docker	17	2	-	-	19
Prometheus	14	-	1	3	18
Etcd	7	12	-	1	20
Go-Ethereum	11	43	6	2	62
TiDB	-	-	-	-	-
gRPC	15	-	1	6	22

Effectiveness of Bug Detection

- 184 previously unknown bugs
  - 170 blocking bugs
  - 14 non-blocking bugs
- 12 FPs due to imprecise static analysis

App	$chan_b$	$select_b$	$range_b$	NBK	Total
Kubernetes	28	4	9	2	43
Docker	17	2	-	-	19
Prometheus	14	-	1	3	18
Etcd	7	12	-	1	20
Go-Ethereum	11	43	6	2	62
TiDB	-	-	-	-	-
gRPC	15	-	1	6	22

Advancement

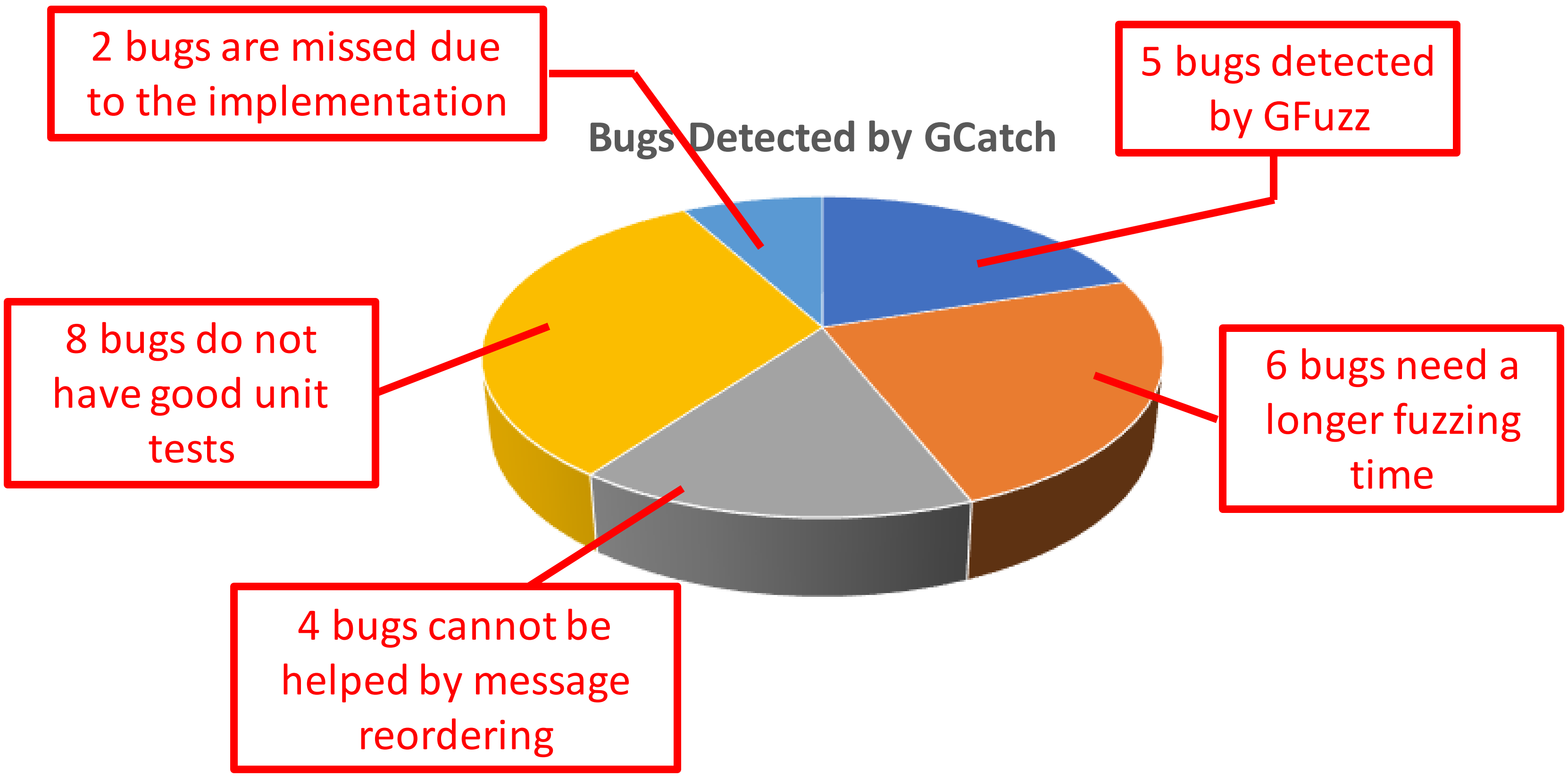
- Compare GFuzz with static detector GCatch
  - Run GFuzz on each application for ***three*** hours
  - Apply GCatch on all packages that can be compiled

85 > 25

App	$GFuzz_3$	GCatch
Kubernetes	18	3
Docker	5	4
Prometheus	8	-
Etcd	7	5
Go-Ethereum	40	5
TiDB	-	-
gRPC	7	8

Bugs Missed by GFuzz

- GFuzz detects 5 bugs captured by GCatch
- GFuzz misses 20 bugs for four reasons



Runtime Overhead

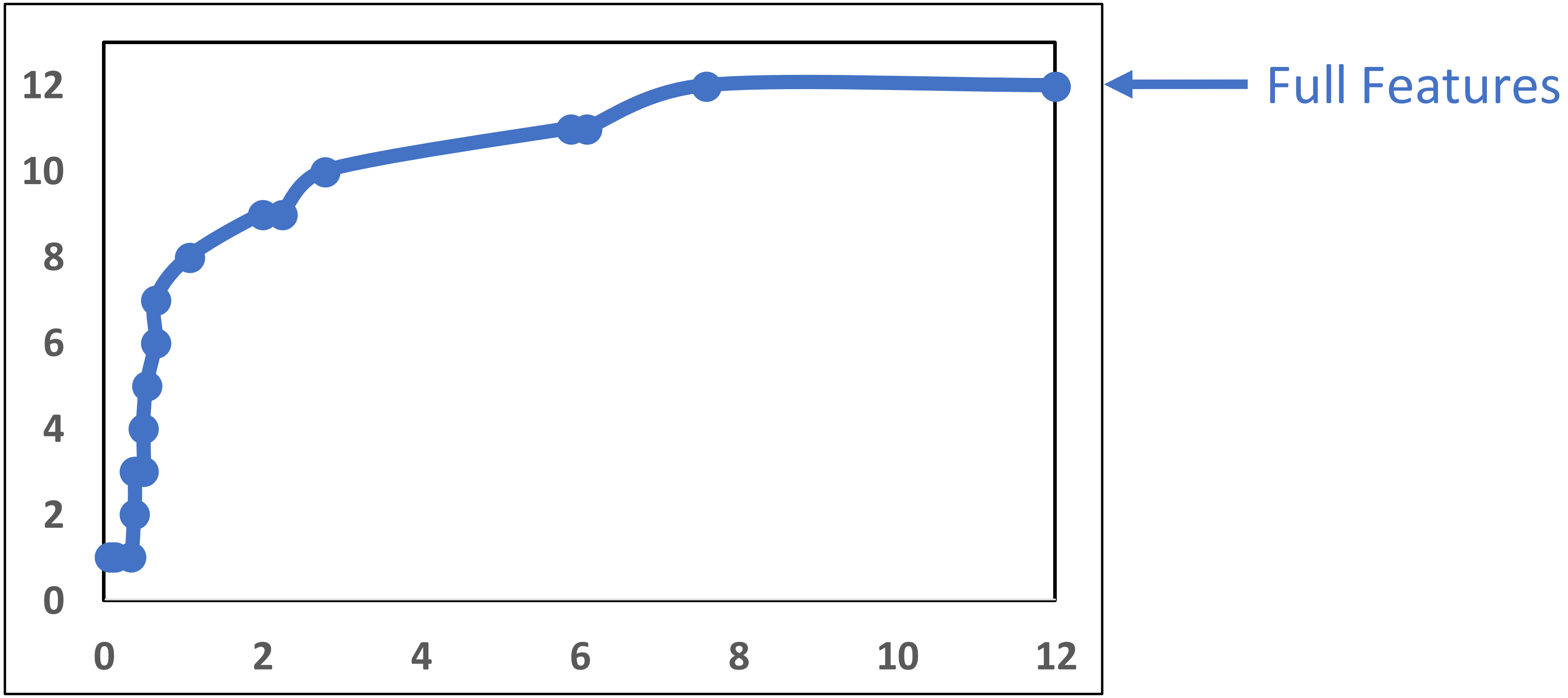
- The average overhead of GFuzz is 3.0X
- The average overhead of the sanitizer is 32.3%

App	$Gfuzz_o$	$Sanitizer_o$
Kubernetes	8.7X	36.65%
Docker	22.7X	44.53%
Prometheus	3.0X	18.08%
Etcd	0.9X	14.43%
Go-Ethereum	20.1X	75.18%
TiDB	1.6X	17.65%
gRPC	8.5X	20.00%

Comparable with ASAN and TSAN

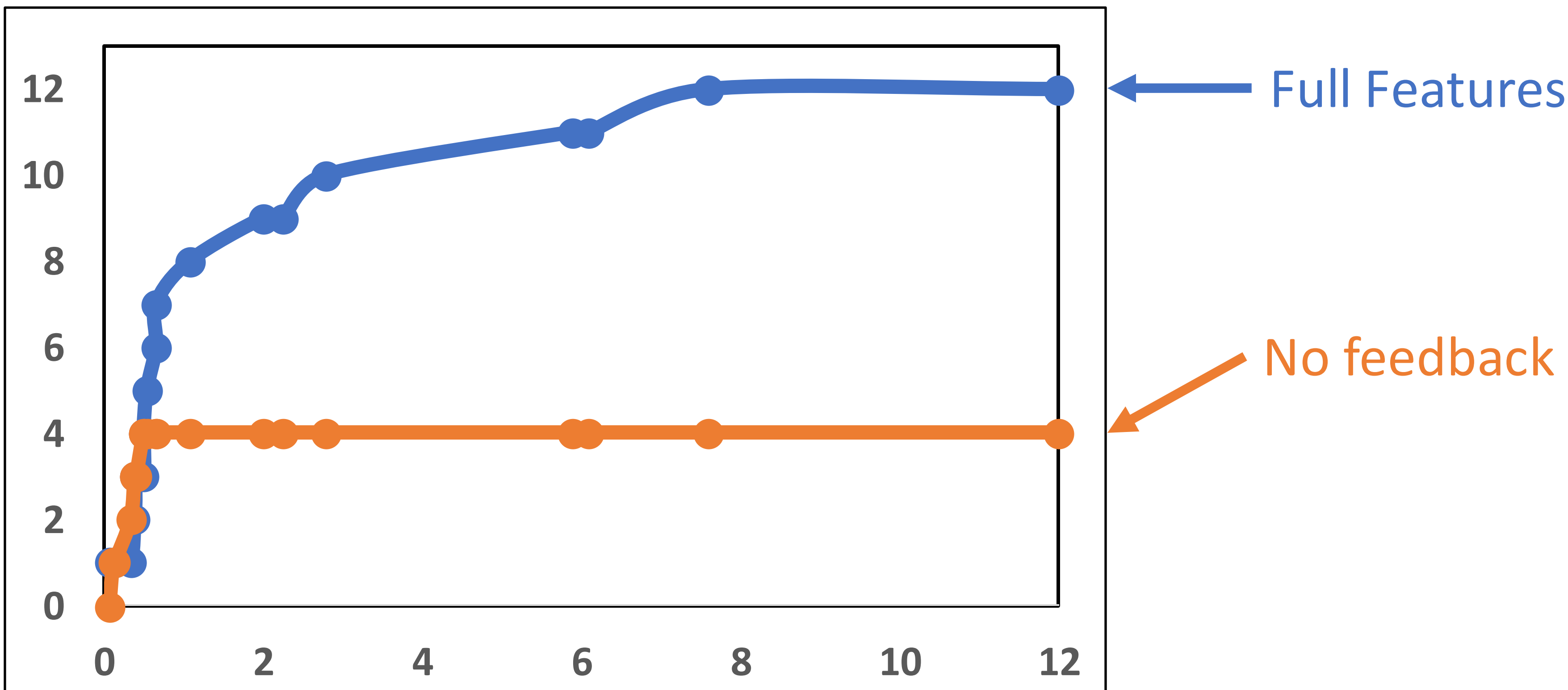
Contribution of Gfuzz's Components

- Apply GFuzz on gRPC using 4 different settings:
  - Enable all components
  - Disable one of the three components



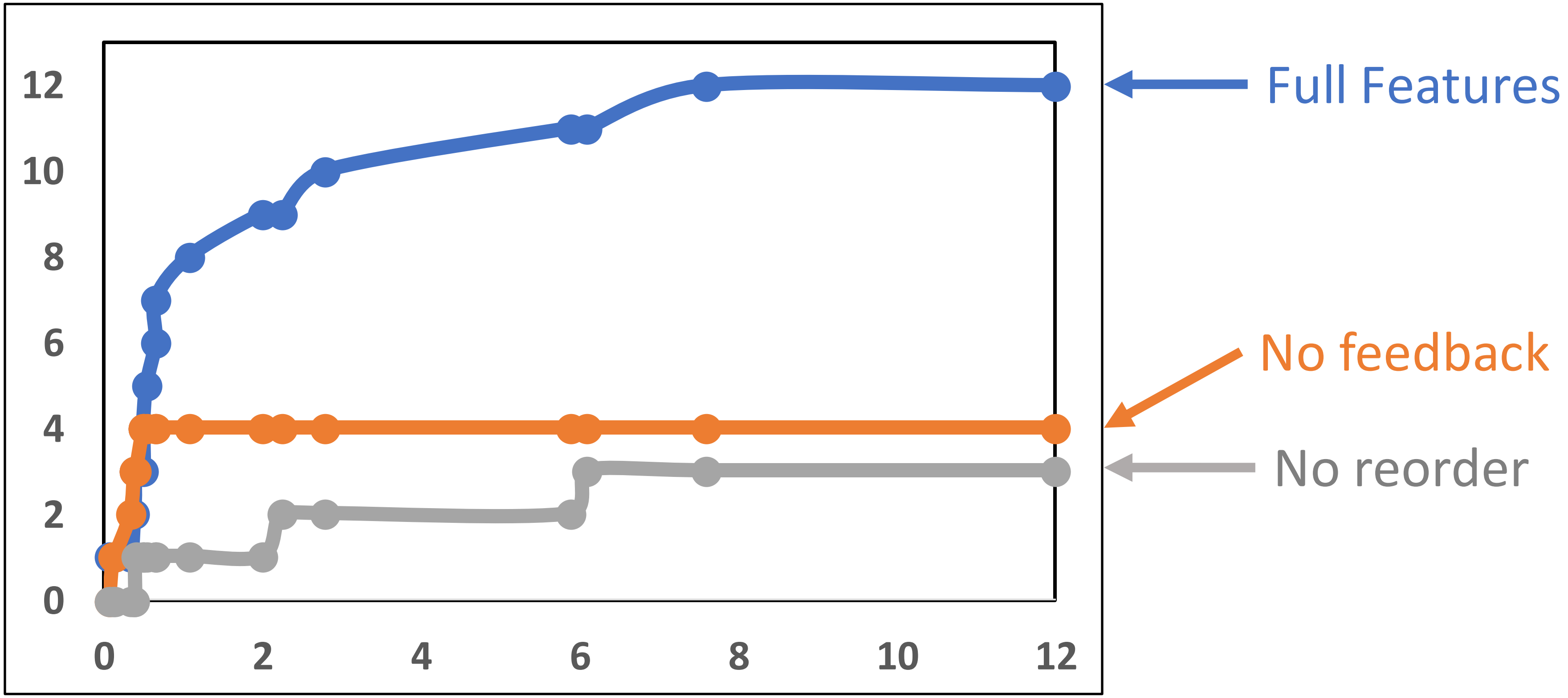
Contribution of Gfuzz's Components

- Apply GFuzz on gRPC using 4 different settings:
  - Enable all components
  - Disable one of the three components



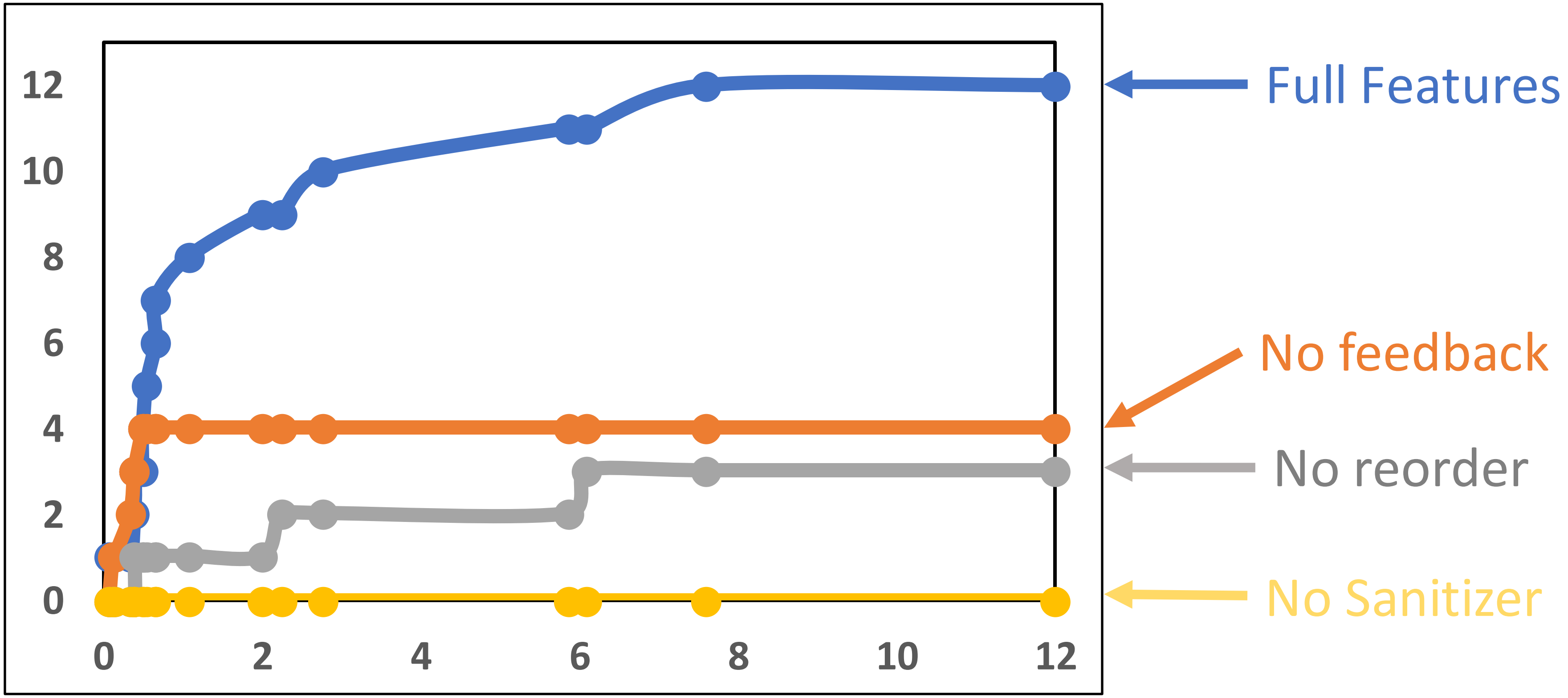
Contribution of Gfuzz's Components

- Apply GFuzz on gRPC using 4 different settings:
  - Enable all components
  - Disable one of the three components



Contribution of Gfuzz's Components

- Apply GFuzz on gRPC using 4 different settings:
  - Enable all components
  - Disable one of the three components



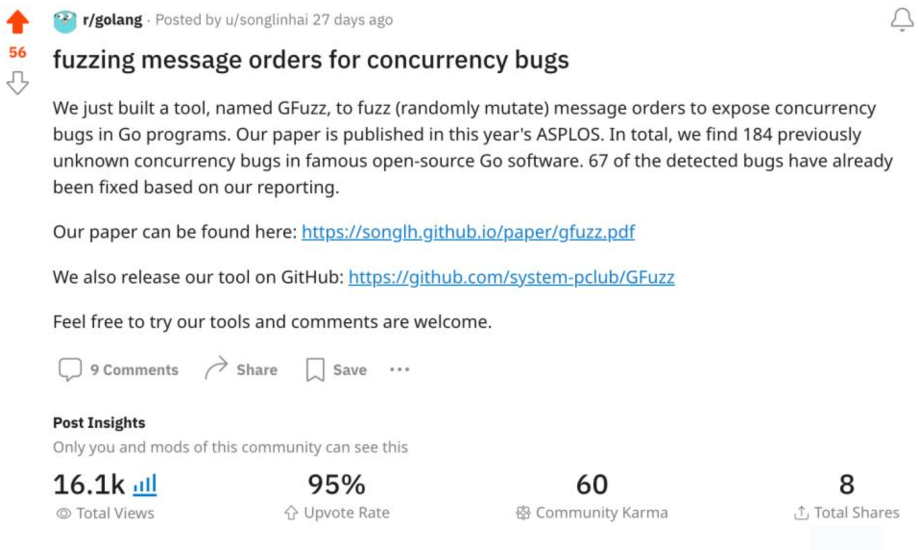
Conclusion

- GFuzz: an effective dynamic bug detector
  - Change message order to explore program states
  - Use feedback to prioritize suspicious orders
  - Propose a sanitizer to capture blocking bugs

  Detected 184 previously unknown bugs in real Go apps

- Future work
  - Integrate other mutation mechanisms
  - Identify more concurrent messages

Thanks a lot!



system-pclub / GFuzz Public

Contact: Shihao Xia (szx5097@psu.edu)

Questions?

- How to identify concurrent messages?
- How to identify suspicious message orders?
- How to capture triggered channel-related bugs?