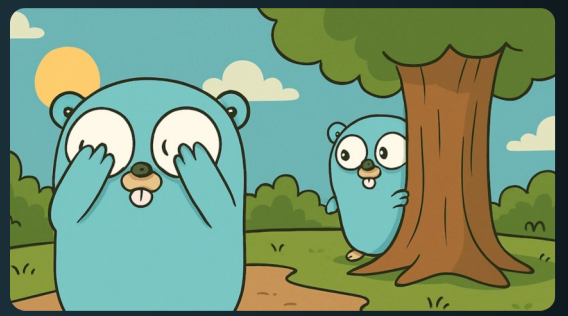


Leak and Seek
A Go Runtime Mystery

Adi Abramovitch, Amit Yahav

Cyolo

Adi Abramovitch

Software Engineer

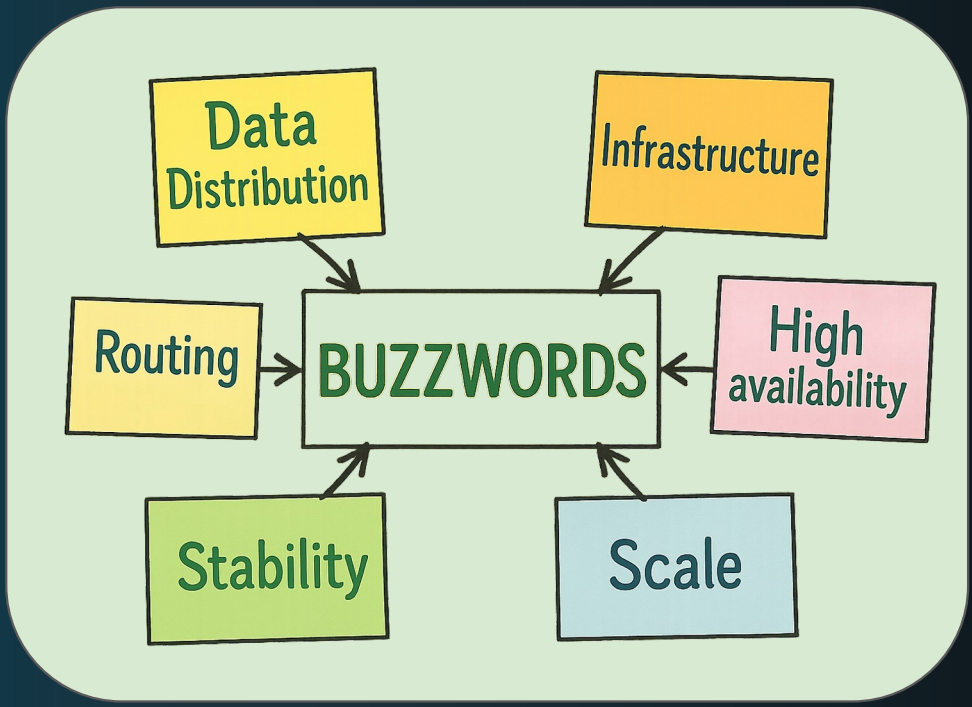
Amit Yahav

Software Engineer

Cyolo

- Cyber Cyber Cyber
- Simplified Secure Remote Access
- Connecting users to critical infrastructure
- Identity-centric and zero-trust architecture

The Platform Team

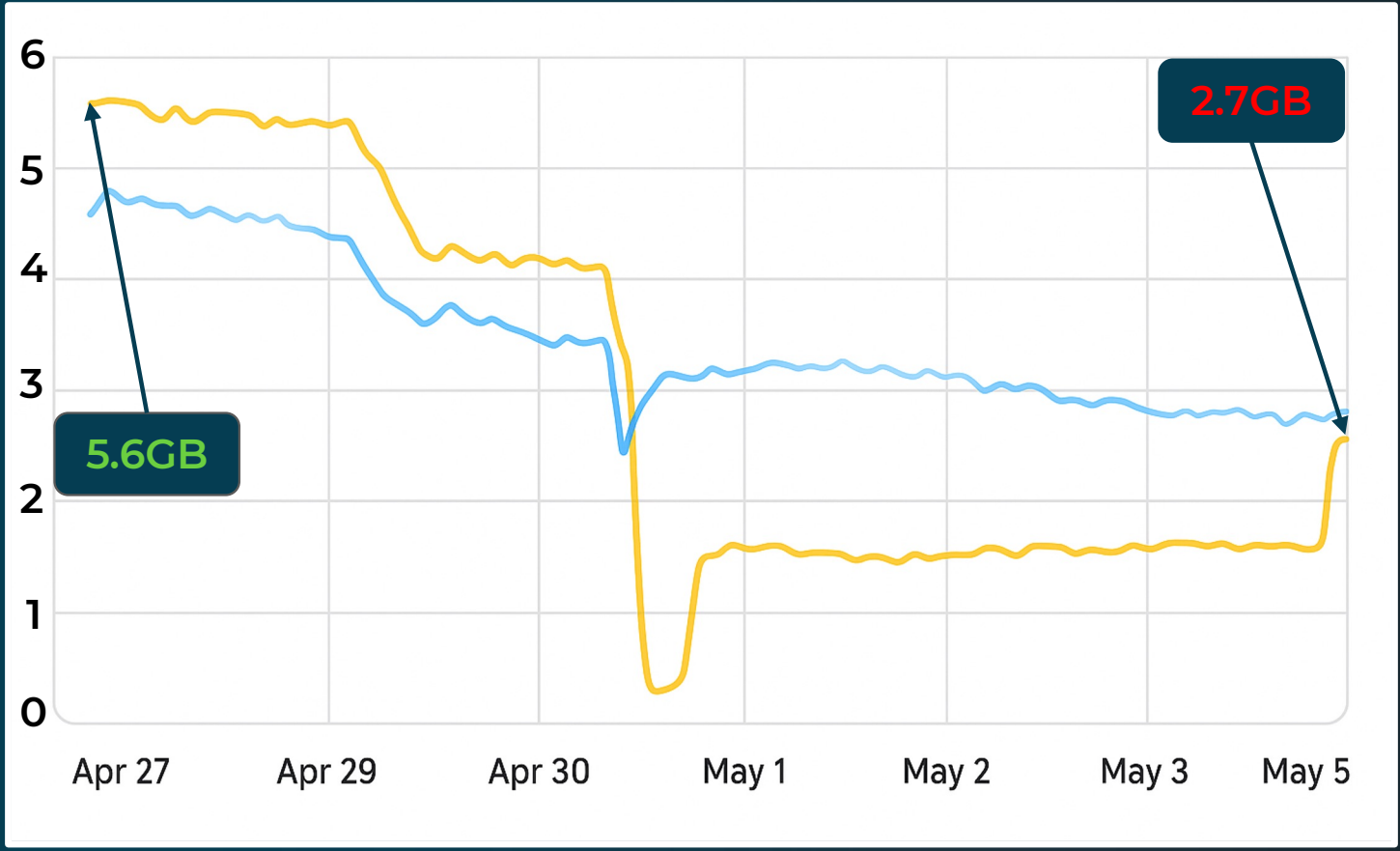


Agenda

1. The **challenge**
2. Common **debugging** techniques
3. How we identified the **root cause**
4. Our **contribution** to Go and our **open-source** solution

Chapter One:
The Alarm Bells

Our Free Memory is Decreasing



What Is a Memory Leak?

"An unintended loss or escape of something"

but...
Go Is a Garbage-Collected Language

Potential Cause: A Goroutine Leak

```go
func main() {
    ch := make(chan byte)

    go func() {
        largeSlice := make([]byte, units.GB)

        // no other routine is pushing data to ch, blocked until the program exists.
        <- ch

        // holds a reference of largeSlice.
        fmt.Printf("slice len: %d", len(largeSlice))
    }()

    // long-running program.
    time.Sleep(60 * time.Minute)
}
```

How Can Go Help Us Detect Leaks?

pprof

What is PPROF?

A tool to diagnose and visualize performance bottlenecks

Available Profiles:

- CPU
- Block
- Mutex
- Goroutines
- Heap

Goroutine Profile

Debug Modes

- **debug=0**
  Compressed binary format, used by `go tool pprof` for visualization

- **debug=1**
  Human-readable, traces are **grouped** together

- **debug=2**
  A full, verbose, human-readable dump of all goroutines

Goroutine Profile with debug=1

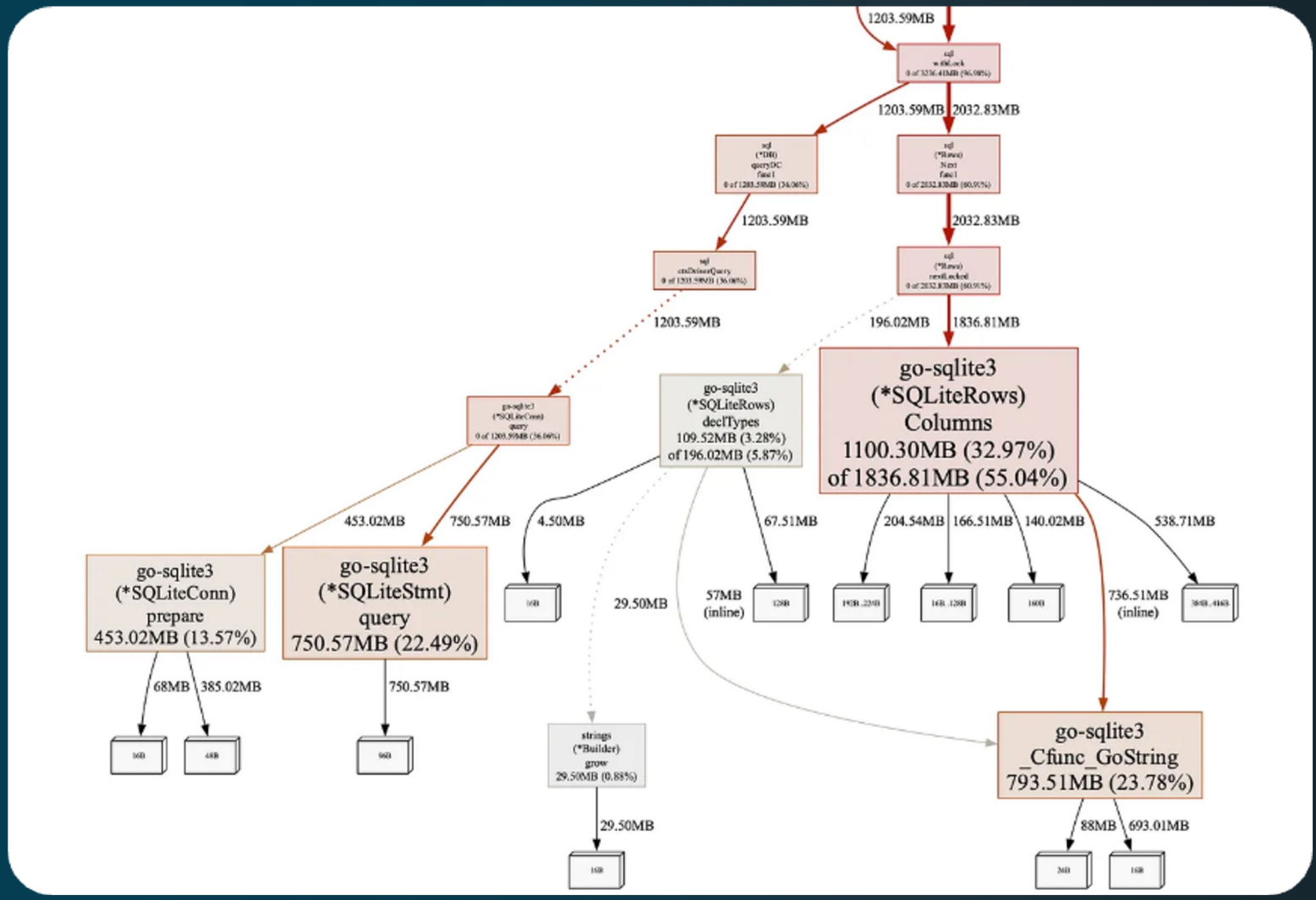
```
goroutine profile: total 957
952 @ 0x10413bc58 0x10411f88c 0x104131394c 0x104143f04
      0x104131394b       main.(*EventEmitter).HandleExcessive+0x14b         /Users/user/go/projects/g/

1 @ 0x1040fae5e 0x10413ab04 0x1042ffb74 0x1042ff990 0x1042fcf28 0x104311e14 0x104312728 0x1042db0f7
      0x1042ffb73       runtime/pprof.writeRuntimeProfile+0xb3             /Users/user/go/pkg/mod/golang.org.
      0x1042ff98f       runtime/pprof.writeGoroutine+0x4f                /Users/user/go/pkg/mod/golang.org.
      0x1042fcf27       runtime/pprof.(*Profile).WriteTo+0x147           /Users/user/go/pkg/mod/golang.org.
      0x104311e13       net/http/pprof.handler.ServeHTTP+0x443         /Users/user/go/pkg/mod/golang.org.
      0x104312727       net/http/pprof.Index+0xc7                      /Users/user/go/pkg/mod/golang.org.
      0x1042db0f7       net/http.HandlerFunc.ServeHTTP+0x37              /Users/user/go/pkg/mod/golang.org.
      0x1042dcef3       net/http.(*ServeMux).ServeHTTP+0x1b3             /Users/user/go/pkg/mod/golang.org.
      0x1042e3d1b       net/http.serverHandler.ServeHTTP+0xbb            /Users/user/go/pkg/mod/golang.org.
      0x1042d9f7b       net/http.(*conn).serve+0x4fb                     /Users/user/go/pkg/mod/golang.org.

1 @ 0x10413bc58 0x1040fe678 0x10413af40 0x104188d18 0x10418982c 0x104213c28 0x10421c8f3 0x10421c8f3 0x1042d503f
      0x10413af3f       internal/poll.runtime_pollWait+0x9f                /Users/user/go/pkg/mod/go:
      0x104188d17       internal/poll.(*pollDesc).wait+0x27                /Users/user/go/pkg/mod/go:
      0x10418982b       internal/poll.(*pollDesc).waitRead+0x1fb               /Users/user/go/pkg/mod/go:
      0x10418981c       internal/poll.(*FD).read+0x1ec                   /Users/user/go/pkg/mod/go:
      0x104213c27       net.(*netFD).Read+0x27                           /Users/user/go/pkg/mod/go:
      0x10421c8f3       net.(*conn).Read+0x33                            /Users/user/go/pkg/mod/go:
      0x1042d503f       net/http.(*connReader).backgroundRead+0x3f       /Users/user/go/pkg/mod/go
```

Goroutine Profile with debug=1

```
goroutine profile: total 957
952 @ 0x10413bc58 0x10411f88c 0x104131394c 0x104143f04
      0x104131394b        main.(*EventEmitter).HandleExcessive+0x14b

1 @ 0x1040fae5e 0x10413ab04 0x1042ffb74 0x1042ff990 0x1042fcf28 0x10
      0x1042ffb73        runtime/pprof.writeRuntimeProfile+0xb3
      0x1042ff98f        runtime/pprof.writeGoroutine+0x4f
      0x1042fcf27        runtime/pprof.(*Profile).WriteTo+0x147
      0x104311e13        net/http/pprof.handler.ServeHTTP+0x443
```

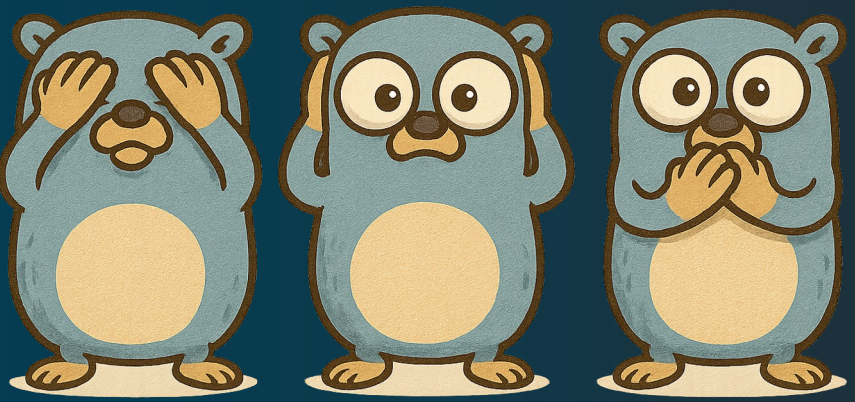
Not in our case

Heap Profile


1203.59MB
1203.59MB
2032.83MB
1203.59MB
2032.83MB
1203.59MB
196.02MB
1836.81MB
go-sqlite3
(*SQLiteRows)
declTypes
109.52MB (3.28%)
of 196.02MB (5.87%)
go-sqlite3
(*SQLiteRows)
Columns
1100.30MB (32.97%)
of 1836.81MB (55.04%)
453.02MB
750.57MB
4.50MB
67.51MB
204.54MB
166.51MB
140.02MB
538.71MB
go-sqlite3
(*SQLiteConn)
prepare
453.02MB (13.57%)
go-sqlite3
(*SQLiteStmt)
query
750.57MB (22.49%)
29.50MB
57MB
(inline)
736.51MB
(inline)
68MB
385.02MB
750.57MB
strings
(*Builder)
grow
29.50MB (0.88%)
go-sqlite3
_Cfunc_GoString
793.51MB (23.78%)
29.50MB
88MB
693.01MB

Chapter Two:

The Three Body Problem



SqliteRows

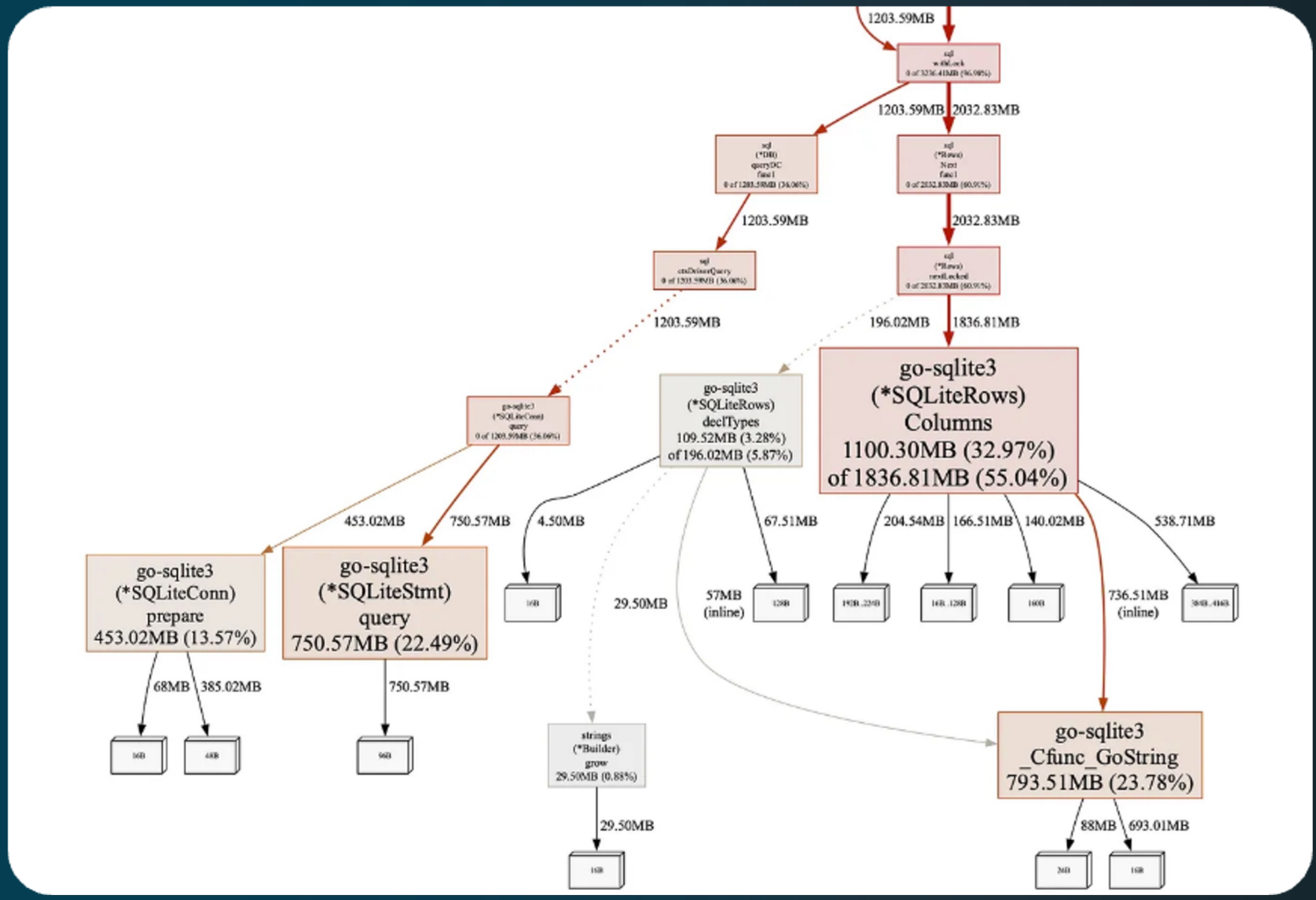
go-sqlite3
(*SQLiteRows)
Columns
1100.30MB (32.97%)

SqliteStmt

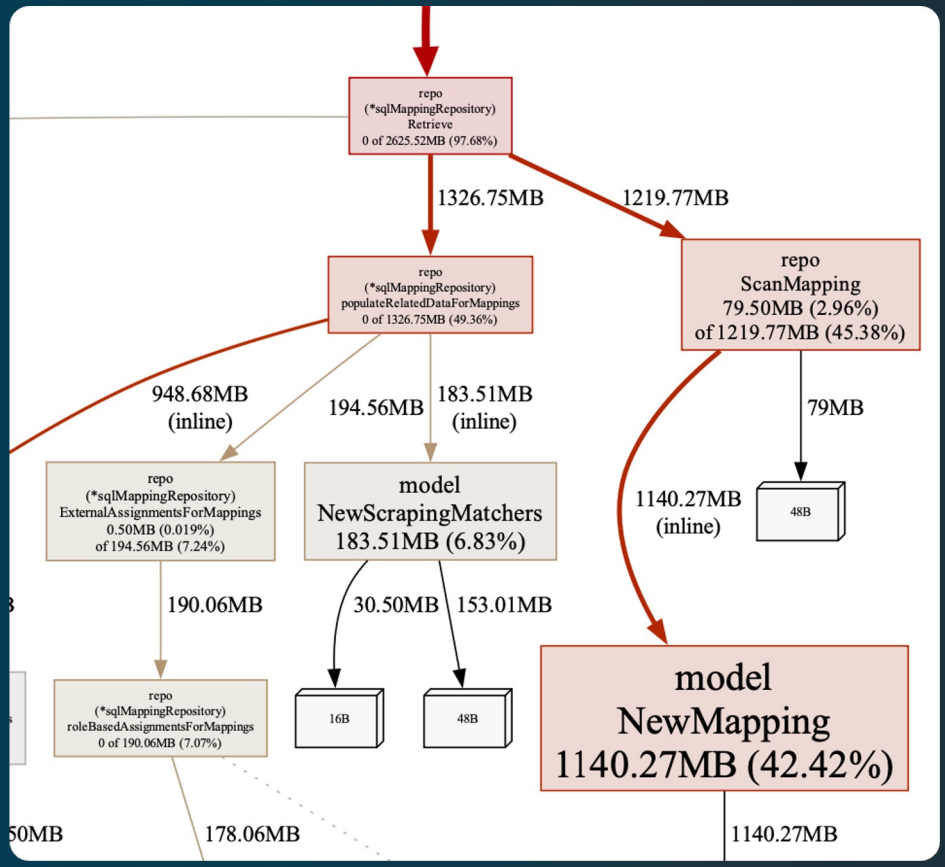
go-sqlite3
(*SQLiteStmt)
query
750.57MB (22.49%)

SqliteConn

go-sqlite3
(*SQLiteConn)
prepare
453.2MB (13.57%)


1203.59MB
1203.59MB
2032.83MB
1203.59MB
2032.83MB
1203.59MB
196.02MB
1836.81MB
go-sqlite3
(*SQLiteRows)
Columns
1100.30MB (32.97%)
of 1836.81MB (55.04%)
go-sqlite3
(*SQLiteRows)
declTypes
109.52MB (3.28%)
of 196.02MB (5.87%)
453.02MB
750.57MB
4.50MB
67.51MB
204.54MB
166.51MB
140.02MB
538.71MB
go-sqlite3
(*SQLiteConn)
prepare
453.02MB (13.57%)
go-sqlite3
(*SQLiteStmt)
query
750.57MB (22.49%)
29.50MB
57MB
(inline)
736.51MB
(inline)
68MB
385.02MB
750.57MB
strings
(*Builder)
grow
29.50MB (0.88%)
go-sqlite3
_Cfunc_GoString
793.51MB (23.78%)
29.50MB
88MB
693.01MB

Simulating the Leak




SHISHI

Suspect No 1: database/sql

- API for relational databases provided by Go
- Maybe it holds onto the references?
- After thorough inspection: No!

Suspect No 2: mattn/go-sqlite3 Driver

Facts:

- SqliteRows → SqliteStmt → SqliteConn
- This library uses CGO
- The version changed
  - Notably: Added **finalizer** to SqliteRows

```go
func (s *SQLiteStmt) query(ctx context.Context, args []driver.NamedValue) (driver.Rows, error)
{   if err := s.bind(args); err != nil {
        return nil, err
    }

    rows := &SQLiteRows{
        s:        s,
        nc:       int(C.sqlite3_column_count(s.s)),
        cols:     nil,
        decltype: nil,
        cls:      s.cls,
        closed:   false,
        ctx:      ctx,
    }
    runtime.SetFinalizer(rows, (*SQLiteRows).Close)

    return rows, il
}
```

What Is a Finalizer?

A function run by the Go runtime on an object right **before it's garbage collected**

```
runtime.SetFinalizer(rows, (*SQLiteRows).Close)
```

Why Use a Finalizer?

Acts as a **last-resort cleanup** in case the user forgets to call Close() or a similar manual cleanup function

Finalizers Are Error Prone

Many examples and scenarios are available online

Sqlite3 got it right!

A Hint in the Docs!

“A single goroutine runs all finalizers for a program, sequentially. If a finalizer must run for a long time, it should do so by starting a new goroutine.”

~runtime/mfinal.go

finq: The Finalizers Queue

```
    finq       *finblock // list of finalizers that are to be executed
```

```
func createfing() {
    // start the finalizer goroutine exactly once
    if fingStatus.Load() == fingUninitialized && fingStatus.CompareAndSwap(fingUninitialized, fingCreated) {
        go runfing()
    }
}
```

~runtime/mfinal.go

Maybe runfinq Is Blocked?

POC: A Blocked Finalizer

- Created a finalizer that will block
- **Bingo!**
  All future objects with an associated finalizer will not be released
- **Easy to verify!**
  The blocked finalizer routine can be seen on the goroutine profile

Goroutine Profile with debug=1

```
goroutine profile: total 6

1 @ 0x104561cb8 0x104541e74 0x104541e55 0x1045630d8 0x10457a670 0x104712324 0x10471231d 0x1047122c5 0x104509a84
0x104569a74
#   0x1045630d7 internal/sync.runtime_SemacquireMutex+0x27  /Users/projects/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.24.1.darwin-arm64/src/runtime/sema.go:95
#   0x10457a66f internal/sync.(*Mutex).lockSlow+0x16f       /Users/projects/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.24.1.darwin-arm64/src/internal/sync/mutex.go:149
#   0x104712323 internal/sync.(*Mutex).Lock+0xb3        /Users/projects/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.24.1.darwin-arm64/src/internal/sync/mutex.go:70
#   0x10471231c sync.(*Mutex).Lock+0xac             /Users/projects/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.24.1.darwin-arm64/src/sync/mutex.go:46
#   0x1047122c4 main.(*Leaky).Close+0x54            /Users/projects/Library/Application
Support/JetBrains/GoLand2025.1/scratches/scratch_76.go:20
#   0x104509a83 runtime.runfinq+0x3d3               /Users/projects/go/pkg/mod/golang.org/toolchain@v0.0.1-
go1.24.1.darwin-arm64/src/runtime/mfinal.go:275
```

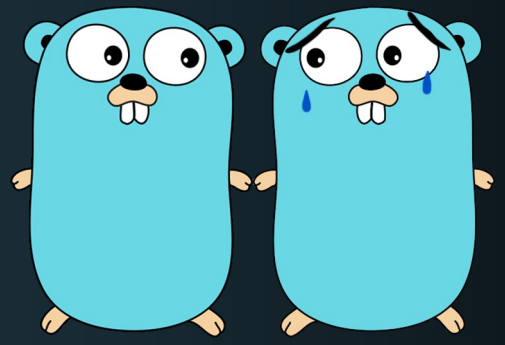
Eureka!

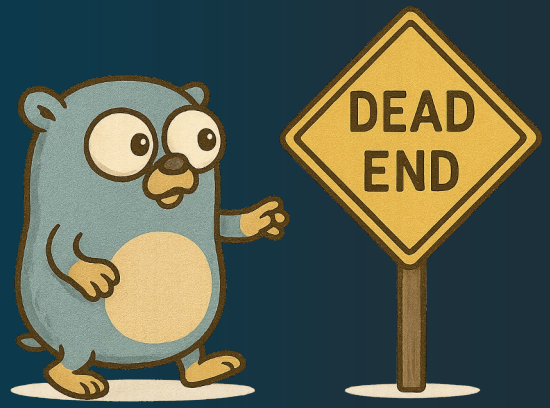
No Trace of the finq Routine...

Conclusion

The blocked finalizer is not the leak's root cause

Dead End?





Chapter Three:

Beyond the Dead-End

Weapons Free: Debug Mode

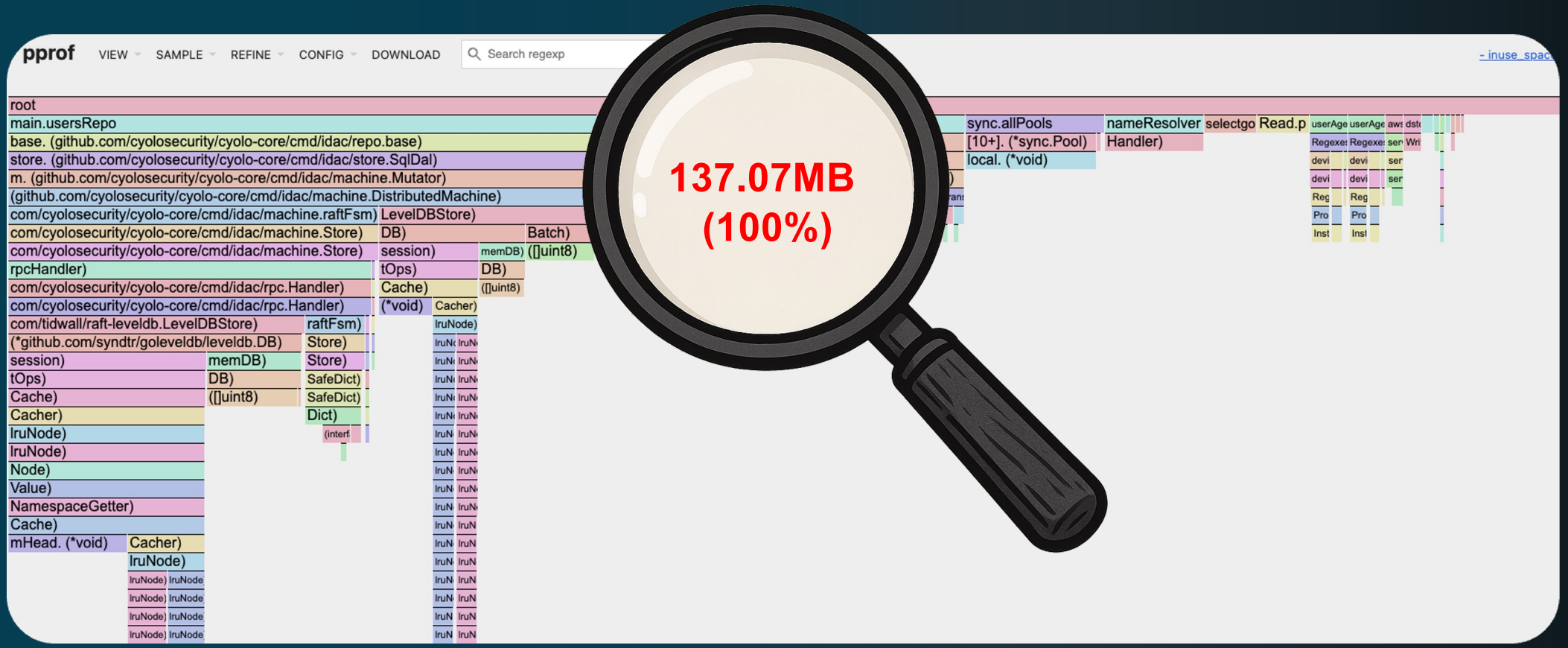
Goref

cloudwego/goref

Weapons Free: Debug Mode

- Build our app in debug mode to utilize the symbol table
- Released a new version to the customer
- Obtained core dump
- Generated goref profile using ***grf core exec_file core_file***
- Observed the output using ***go tool pprof -http=:5079 ./grf.out***

go tool pprof -http=:5079 ./grf.out



What Do We Know So Far

- We identified the **flows** that lead to the leaking allocation
- We identified an additional **finalizer** in sqlite3
- We recreated a leak using a **blocked** finalizer
- We retrieved a goroutine profile from the **customer**
- **No trace** for runfinq in the goroutine profile

Conclusion, yet again...

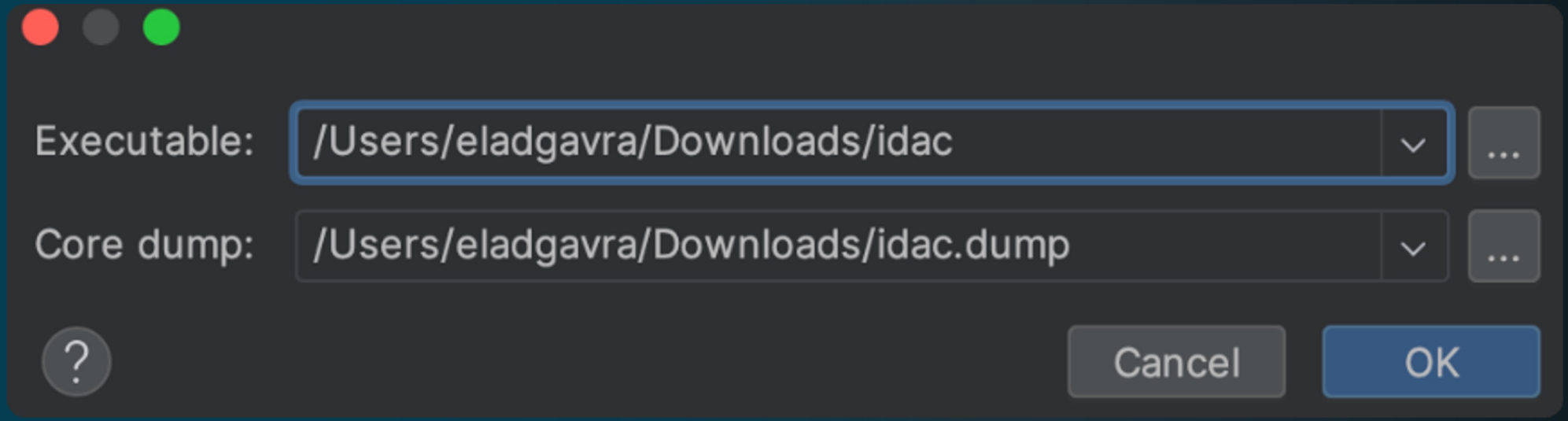
The blocked finalizer is not the leak's root cause

Doubting Ourselves

- A Go runtime bug?
- Goroutine profile is not accurate?
- A blocked finalizer?

Analyzing the Core Dump

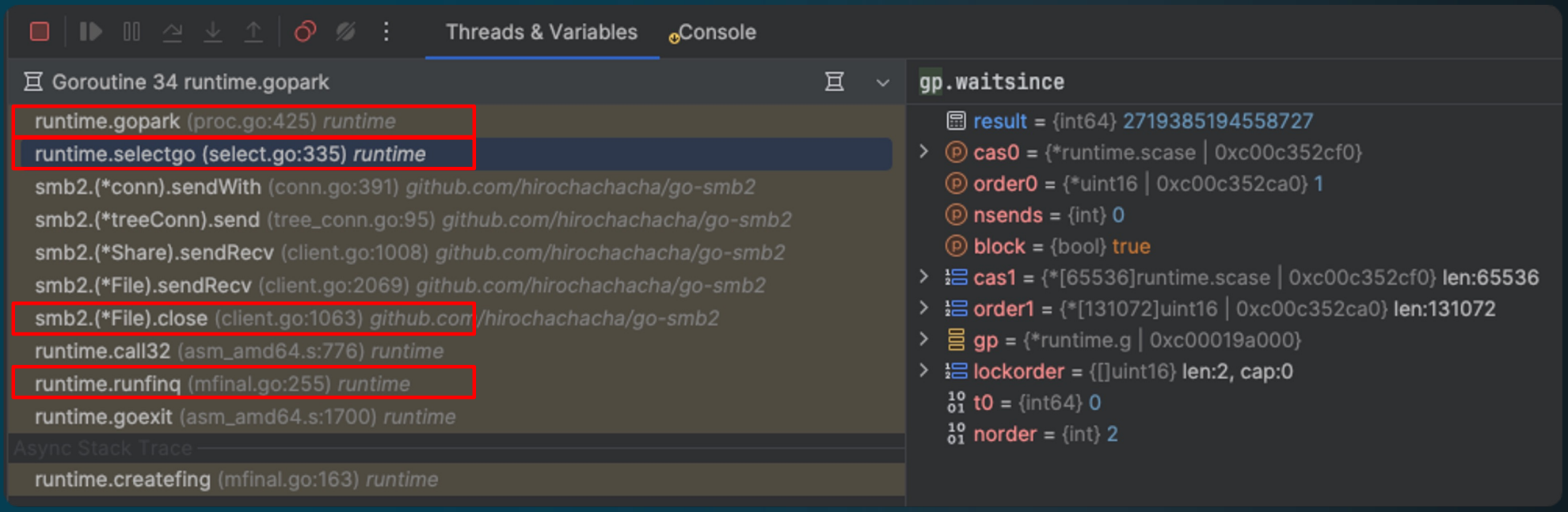
- GoLand > Run > Open Core Dump



Our Next Objectives

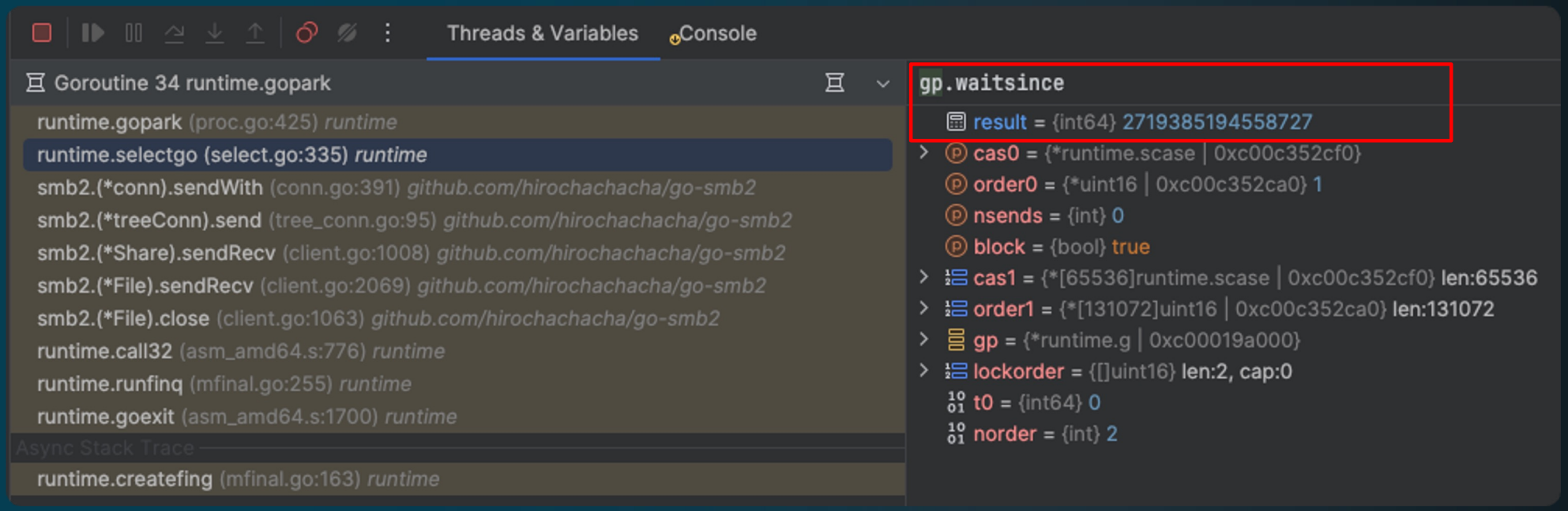
- Trace of runfinq's routine
- What is it currently doing?
- Is it blocked?

The Infamous: Goroutine 34



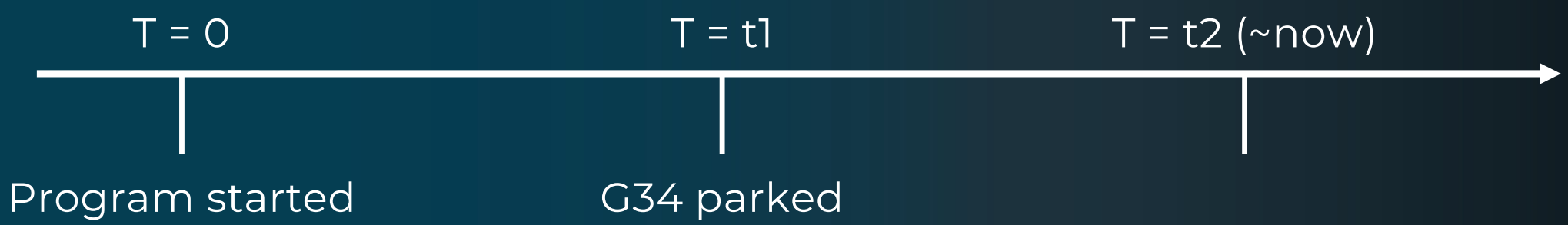
The Infamous: Goroutine 34

gp.waitsince - the nanosecond timestamp of a goroutine's entry into a waiting state (parked).



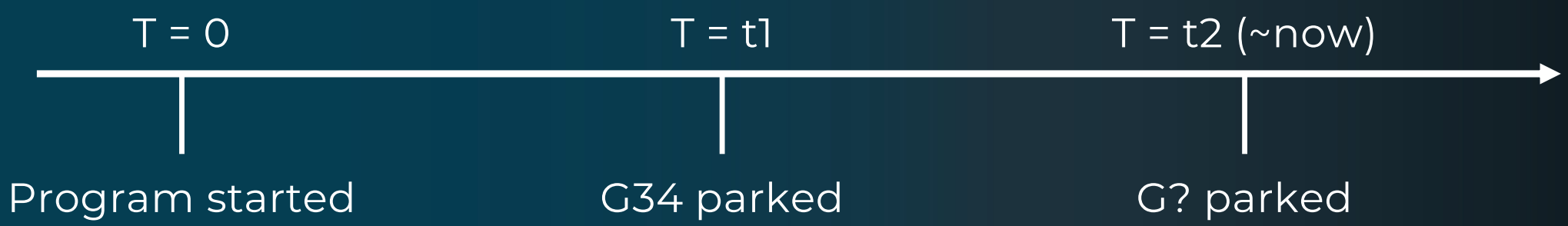
How long has G34 been parked?

t2 - t1 = ?



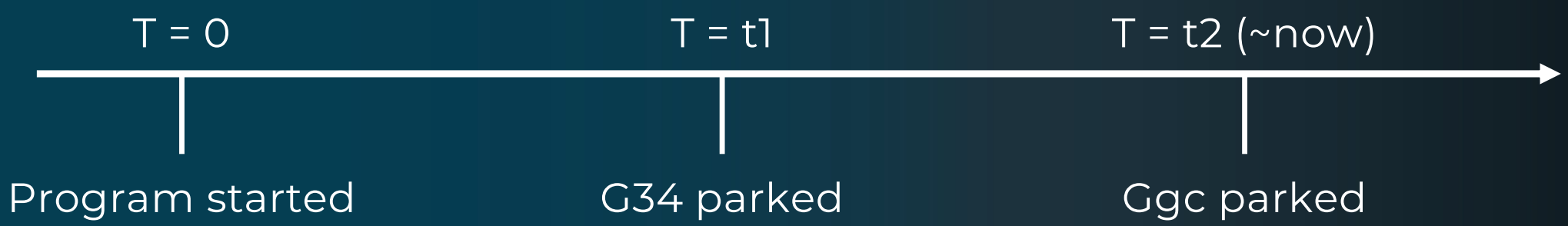
How long has G34 been parked?

t2 - t1 = ?



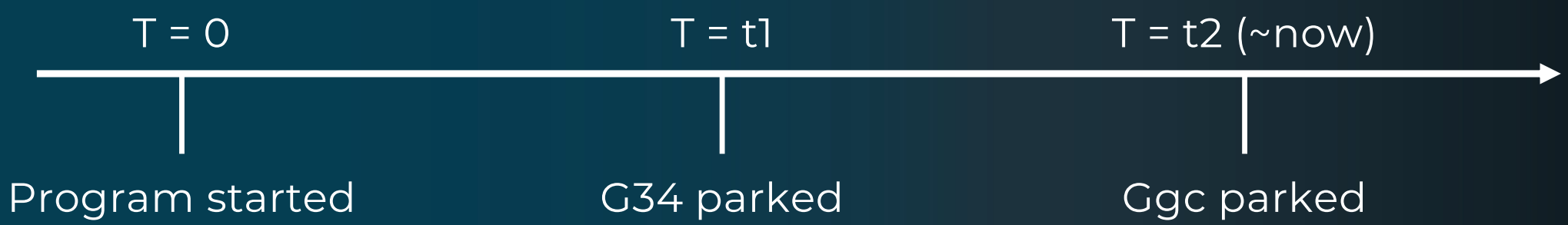
How long has G34 been parked?

t2 - t1 = ?



How long has G34 been parked?

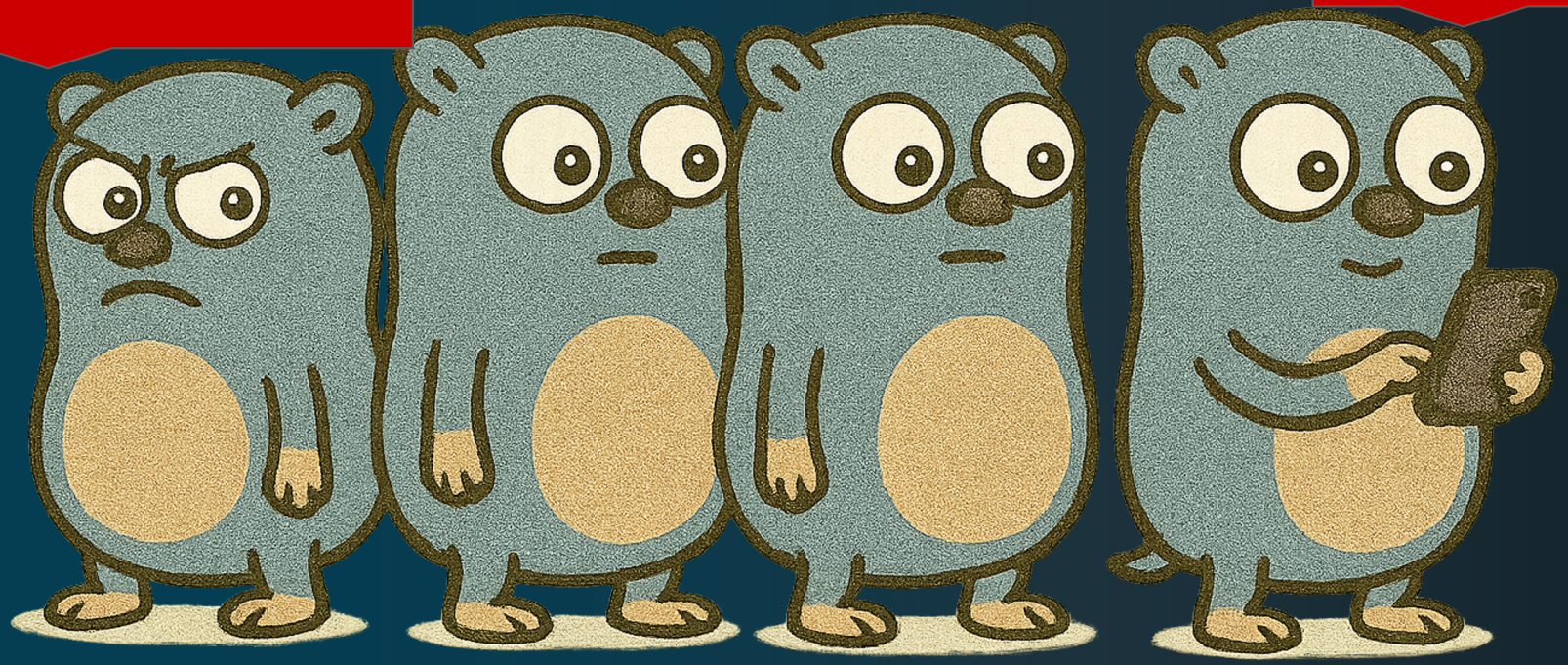
t2 - t1 = 197 minutes



Finalizers Queue

(*SqliteRows).Close

smb2.(*File).close



hirochachacha/go-smb2

```go
func (fs *Share) newFile(r CreateResponseDecoder, name string) *File {
    fd := r.FileId().Decode()

    fileStat := &FileStat{
        CreationTime:   time.Unix(0, r.CreationTime().Nanoseconds()),
        LastAccessTime: time.Unix(0, r.LastAccessTime().Nanoseconds()),
        LastWriteTime:  time.Unix(0, r.LastWriteTime().Nanoseconds()),
        ChangeTime:     time.Unix(0, r.ChangeTime().Nanoseconds()),
        EndOfFile:      r.EndofFile(),
        AllocationSize: r.AllocationSize(),
        FileAttributes: r.FileAttributes(),
        FileName:       base(name),
    }

    f := &File{fs: fs, fd: fd, name: name, fileStat: fileStat}

    runtime.SetFinalizer(f, (*File).close)

    return f
}
```

Issues Identified

- (*File).close did not run in a separate goroutine
- **197 minutes** for an I/O operation is not reasonable
- SMB finalizer is blocked

Why SMB Finalizer is Blocked?

Race condition in **(*File).close** method

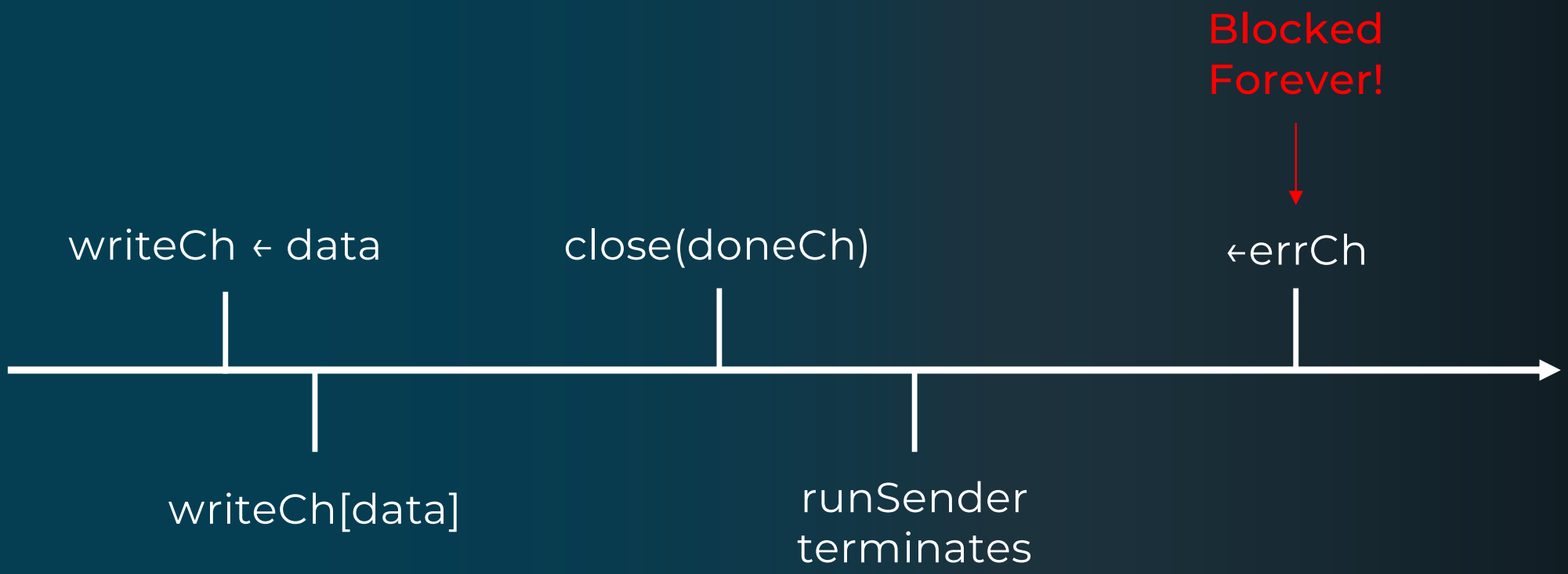
The Race

*Finalizer: (*File).close*

```
...
select {
case conn.writeCh <- rr.pkt:
    select {
    case err = <-conn.errCh:
        ...
    case <-ctx.Done():
        return
    }
case <-ctx.Done():
    ...
}
...
```

```
func (conn *conn) runSender() {
    for {
        select {
        case <-conn.doneCh:
            return
        case pkt := <-conn.writeCh:
            _, err := conn.t.Write(pkt)

            conn.errCh <- err
        }
    }
}
```

The Race


Blocked
Forever!
writeCh ← data
close(doneCh)
←errCh
writeCh[data]
runSender
terminates

The Race

*Finalizer: (*File).close*

```
...
select {
case conn.writeCh <- rr.pkt:
    select {
    case err = <-conn.errCh:
        ...
    case <-ctx.Done():
        return
    }
case <-ctx.Done():
    ...
}
...
```

```
func (conn *conn) runSender() {
    for {
        select {
        case <-conn.doneCh:
            return
        case pkt := <-conn.writeCh:
            _, err := conn.t.Write(pkt)

            conn.errCh <- err
        }
    }
}
```

The Fix: Bypassing The Race

```
ctx, cancel := context.WithCancel(parentCtx)
defer cancel()
...
```

The Race

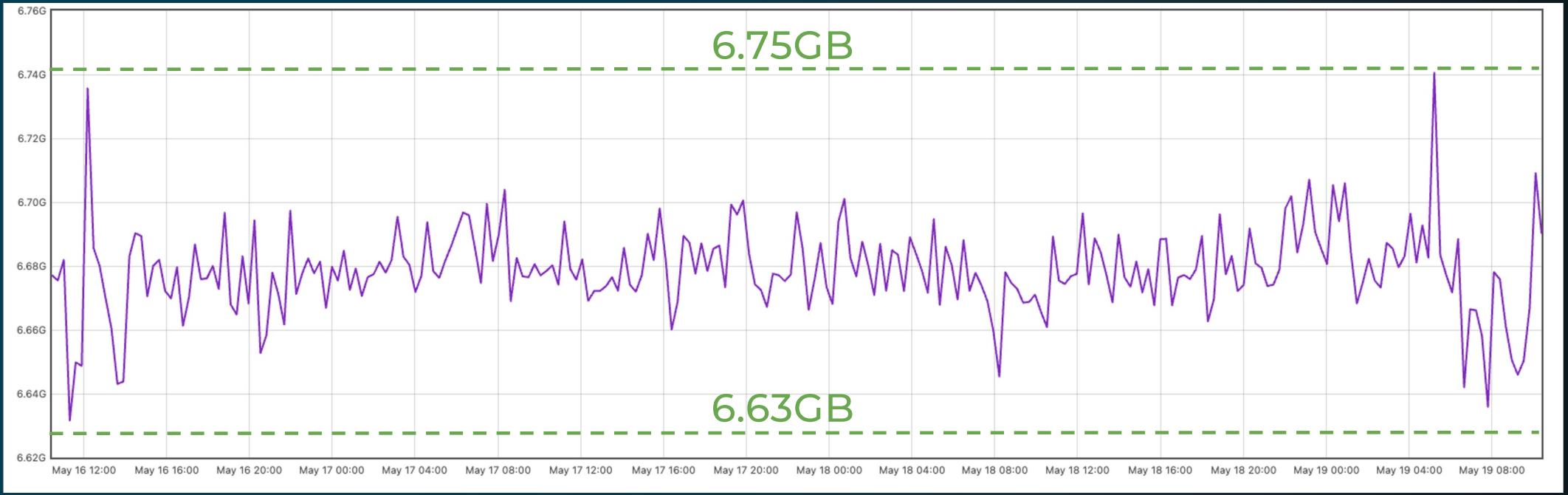
*Finalizer: (*File).close*

```
...
select {
case conn.writeCh <- rr.pkt:
    select {
    case err = <-conn.errCh:
        ...
    case <-ctx.Done():
        return
    }
case <-ctx.Done():
    ...
}
...
```

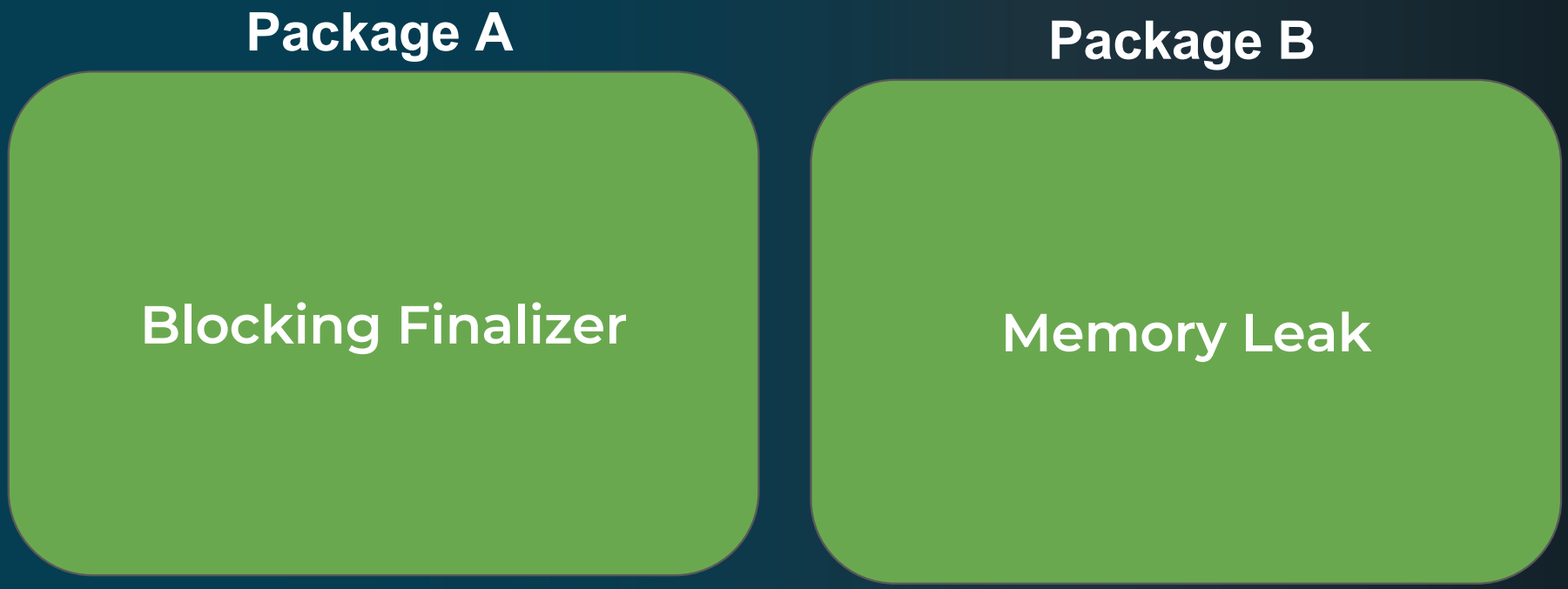
```
func (conn *conn) runSender() {
    for {
        select {
        case <-conn.doneCh:
            return
        case pkt := <-conn.writeCh:
            _, err := conn.t.Write(pkt)

            conn.errCh <- err
        }
    }
}
```

Solved!

Free Memory


6.75GB
6.63GB
6.76G
6.74G
6.72G
6.70G
6.68G
6.66G
6.64G
6.62G
May 16 12:00
May 16 16:00
May 16 20:00
May 17 00:00
May 17 04:00
May 17 08:00
May 17 12:00
May 17 16:00
May 17 20:00
May 18 00:00
May 18 04:00
May 18 08:00
May 18 12:00
May 18 16:00
May 18 20:00
May 19 00:00
May 19 04:00
May 19 08:00

The Implication



Epilogue:

From Bug to Better Go

Open Questions

- Why Goref did not capture our leak?
- Why was runfinq missing from the goroutine profile?

Why Goref Did not Capture Our Leak?

```
// finblock is allocated from non-GC'd memory, so any heap pointers
// must be specially handled. GC currently assumes that the finalizer
// queue does not grow during marking (but it can shrink).
type finblock struct {
    _       sys.NotInHeap
    alllink *finblock
    next    *finblock
    cnt     uint32
    _       int32
    fin     [(_FinBlockSize - 2*goarch.PtrSize - 2*4) / unsafe.Sizeof(finalizer{})]finalizer
}
```

Why Was runfinq Missing From The Goroutine Profile?

- We recreated the blocked routine scenario
- ‘runfinq’ is visible with **debug=1**
- ‘runfinq’ is not visible with **debug=2**

A bug!

Follow-up Go Issues

- **[#73011]** Goroutine profile at debug=2 hides the start of the finalizer goroutine. Fixed in Go 1.25!
- **[#72950]** Explicit documentation for blocked finalizers and their consequences
- **[#72948]** A metric for detecting slow or stalled finalizer queues

- **[#72949]** A dedicated debug mode for finalizers

Are we done? Not yet!

We need to detect the next blocking finalizer,

effortlessly, starting **yesterday**.

Implementing a Proactive Monitoring Solution

1. Generate a **disposable** object with an associated finalizer

2. Measure the **duration** from enqueuing until execution

3. Export this data as a **metric**

4. Upon blockage: **notify** with a trace of the blocking finalizer

Metric: Finalizer Duration

Introducing: finq

A tool for monitoring Go's finalizer routine responsiveness.

```go
go finq.Monitor(ctx, &MonitorOpts{
    StallingInterval: time.Minute,
    OnStalling: func(d time.Duration) {
        log.Println("waiting for finalizer to execute for:", d)

        // you can use this functionality to retrieve the stacktrace of the blocking routine
        trace := finq.GetFinalizerStackTrace()
        if trace != "" {
            log.Println("blocking routine trace:", trace)
        }
    },
    OnComplete: func(d time.Duration) {
        log.Println("finalizer executed after:", d)
    },
    ImmediatelyTriggerGC: false,
})
```

github.com/cyolosecurity/finq

AddCleanup in Go 1.24

```
    cleanupFunc := func(fileName string) {
            if err := os.Remove(fileName); err == nil {
                fmt.Println("temp file has been removed")
            }
        }

    runtime.AddCleanup(&tempFile, cleanupFunc, tempFile.Name())
```

AddCleanup in Go 1.24

AddCleanup is much less error-prone than SetFinalizer

Would it have solved our problem?

No

Summary

Unusual Leak

Debugging Arsenal

Contribution

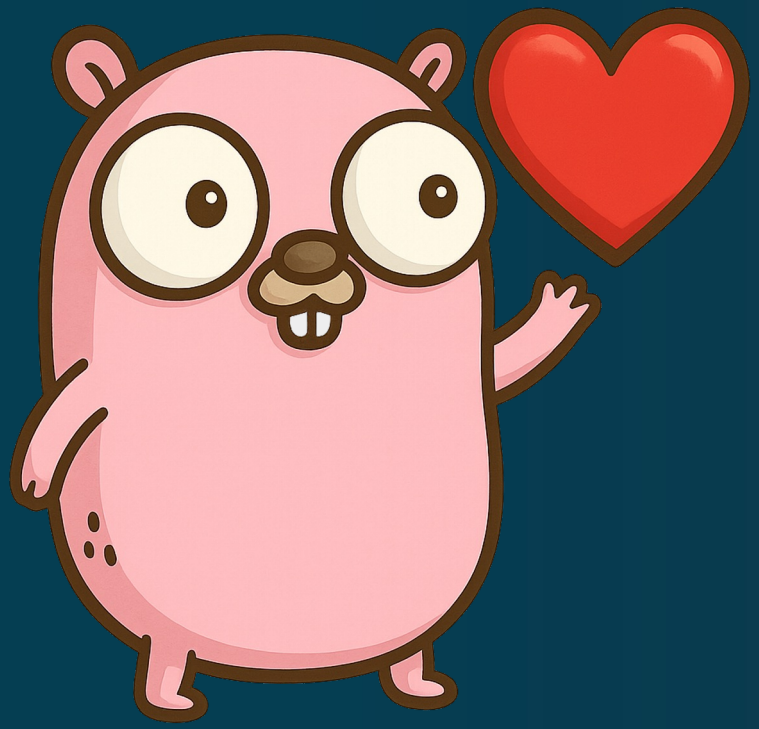
Our Key Takeaways

Persistency

Collaboration

Solidarity

Thank You!



Questions?