Fuzzers, analyzers, and other **Gopher insecticides**

whoami

- alex `hex0punk` useche
- Lead security engineer @ Trail of Bits
- Former software developer consultant

So, what are we learning today?

- Common bugs in Go code
- Fun edge cases
- Focus on concurrency bugs
- Techniques for catching them
  - Understand how they work
  - AST parsing vs. SSA analysis tools
  - How to choose your weapon
- Evaluate the state of security tooling for Go

Why?

- Demystify how SCA tools work
- Core concepts to understanding how static analysis tools work in Go
- Helps us better compare tooling
  - Know what we can get from one tool that we cannot get from the other

Why Go?

An ode to Go

- Garbage collected
- Strongly typed
- Makes concurrency easy
- Composition instead of inheritance
- Easy to read, easy to write
- Easy to fuzz

Common Go bugs

The comedy of Go errors

```go
_, err := verifyAccess()
g, err := getProfile()
if err != nil {
    return err
}

if myVal, err := foo(); myVal != 0 {
    // code code code
}
```

The comedy of Go errors

- Error handling sucks in Go
- Awfully manual
- Copy/paste logic that can lead to mistakes, logic vulnerabilities

Interface weirdness (nil-not-nil)

```go
func main() {
    fmt.Println("file path: ")
    var filePath string
    fmt.Scanln(&filePath)

    err := PrintFile(filePath)
    if err != nil {
        myErr := err.Error() // Panic here
        fmt.Println(myErr)
    }
}
```

```go
func PrintFile(filePath string) error {
    var pathError *os.PathError

    _, err := os.Stat(filePath)
    if err != nil {
        pathError = &os.PathError{
            Path: filePath,
            Err: errors.New("File not found"),
        }
        return pathError
    }
    content, _ := os.ReadFile(filePath)
    fmt.Println(string(content))

    return pathError
}
```

Checkout my blog for low level details

Integers

```
v, err := strconv.Atoi("4294967377")
        g := int32(v)
        fmt.Printf("v: %v, g: %v\n", v, g);
```

Playground

Integers [1]

```go
func main() {
    res := dumbParse("E2147483650")
    fmt.Println("Result: ", res)

}
```

```go
func dumbParse(s string) int32 {
    if len(s) > 1 && (s[0] == 'E' || s[0] == 'e') {
        parsed, err := strconv.ParseInt(string(s[1:]), 10, 64)
        if err != nil {
            return -1
        }
        return int32(parsed)
    }
    return 0
}
```

[1] Playground

Slices are just views. They point to an array.

```go
func main() {
    a := []byte("bag")

    s1 := a[:]
    s2 := a[:]

    s1[2] = 'd'
    s2[2] = 't'


    fmt.Println(string(s1))
    fmt.Println(string(s2))
}
```

Playground

Go concurrency

The problem with go concurrency

— Writing concurrent code in Go is straightforward to do (for most use cases)

— So easy that sometimes concurrency is overused

— It is also easy to make mistakes that can lead to bugs

Taxonomy of go concurrency bugs

Category	Blocking	Non-Blocking
Misuse of channels (message passing)	Goroutine leaks, improper use of context	Closing channels more than once, misuse of `select/case` blocks
Misuse of shared memory	Deadlocks, misuse of `Wait()`	Data races, slice bugs, misuse of anonymous functions, misuse of `WaitGroup`, misuse of special libraries

Common data races [2]

```go
func ConcurrentFunctions(fns ...func()) {
    var wg sync.WaitGroup
    for _, fn := range fns {
        wg.Add(1)
        go func() {
            fn()
            wg.Done()
        }()
    }

    wg.Wait()
}
```

```go
func main() {
    ConcurrentFunctions(func1, func2)
}

func func1() {
    fmt.Println("I am function func1")
}

func func2() {
    fmt.Println("I am function func2")
}
```

[2] Playground

Misuse of channels

Go Channels

Channels

```go
func fibonacci(n int, c chan int) {
	x, y := 0, 1
	for i := 0; i < n; i++ {
		c <- x
		x, y = y, x+y
	}
	close(c)
}
```

```go
func main() {
	c := make(chan int, 10)
	go fibonacci(cap(c), c)
	for i := range c {
		fmt.Println(i)
	}
}
```

Channels & Select [3]

```go
func main() {
    finishReq(1)
    time.Sleep(time.Second * 5)
    fmt.Println(result)
    fmt.Println(runtime.NumGoroutine())
}

func test() string {
    time.Sleep(time.Second * 2)
    return "very important data"
}
```

```go
func finishReq(timeout time.Duration) string {
    ch := make(chan string,2)

    go func() {
        newData := test()
        ch <- newData // block
    }()
    go func() {
        newData := test()
        ch <- newData // block
    }()
    select {
    case result = <- ch:
        fmt.Println("case result")
        return result
    case <- time.After(timeout):
        fmt.Println("case time.Afer")
            return ""
    }
}
```

[3] playground

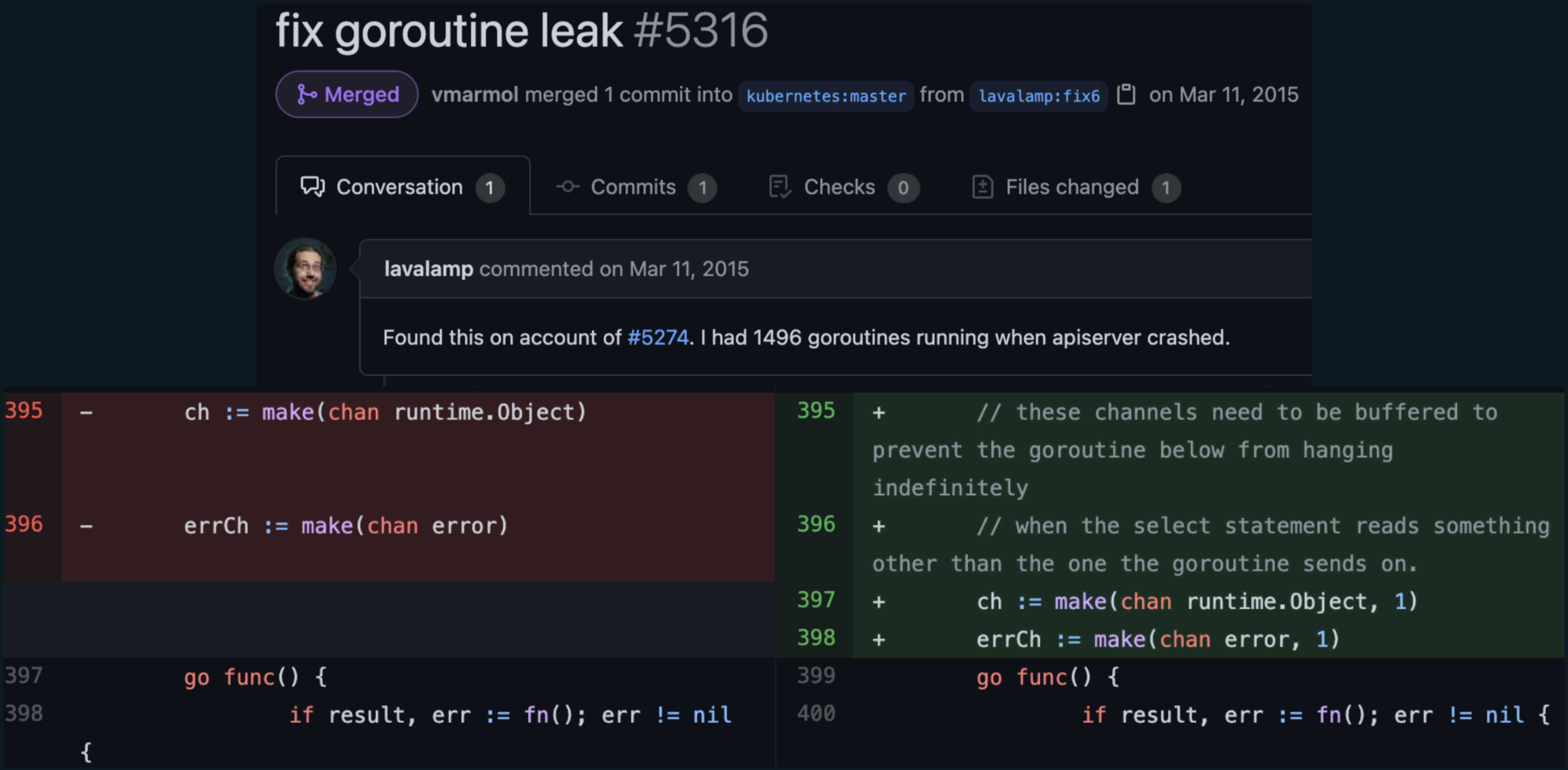
Goroutine Leaks [4]

```
func finishReq(timeout time.Duration) string {
    ch := make(chan string)

    go func() {
        newData := test()
        ch <- newData // block
    }()
    select {
    case result = <- ch:
        fmt.Println("case result")
        return result
    case <- time.After(timeout):
        fmt.Println("case time.After")
            return ""
    }
}
```

[4] Playground

So what?



fix goroutine leak #5316
Merged
vmarmol merged 1 commit into kubernetes:master from lavalamp:fix6 on Mar 11, 2015
Conversation 1
Commits 1
Checks 0
Files changed 1
lavalamp commented on Mar 11, 2015
Found this on account of #5274. I had 1496 goroutines running when apiserver crashed.
395 - ch := make(chan runtime.Object)
396 - errCh := make(chan error)
395 + // these channels need to be buffered to prevent the goroutine below from hanging indefinitely
396 + // when the select statement reads something other than the one the goroutine sends on.
397 + ch := make(chan runtime.Object, 1)
398 + errCh := make(chan error, 1)
397 go func() {
398 if result, err := fn(); err != nil {
399 go func() {
400 if result, err := fn(); err != nil {

Missing Unlocks [6]

```go
func (ms *myStruct) evalEvenNumbers() error {
    fmt.Println("locking ms with num: ", ms.num)

    ms.lock.Lock()

    if ms.num % 2 != 0  {
        fmt.Println("no longer even")
        return fmt.Errorf("invalid")
    }

    ms.num += 2

    ms.lock.Unlock()

    return nil
}
```

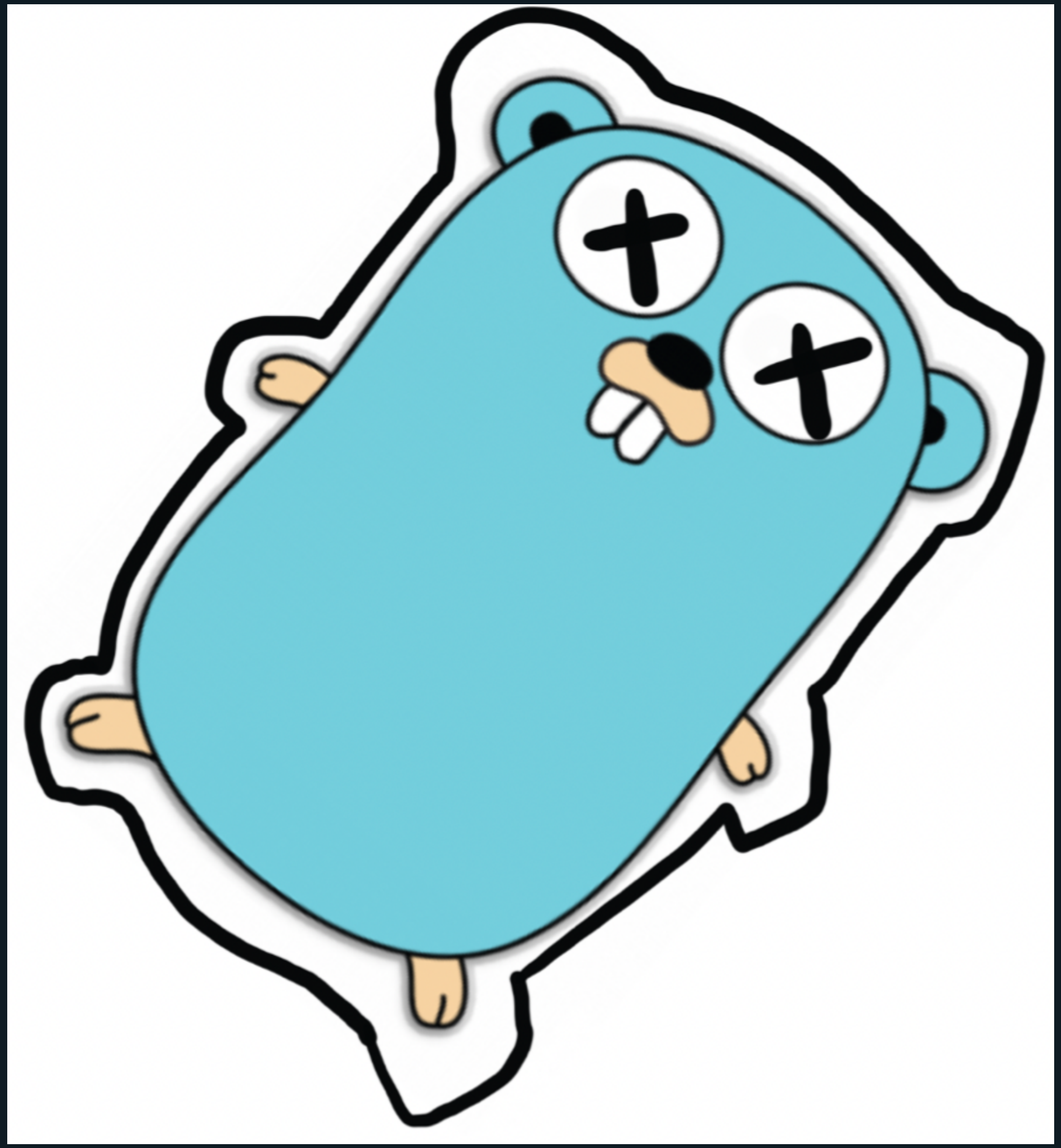
[6] playground

RLock double locks [7]

"If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released."

[7] playgorund

This has been terrifying. Now what?



The insecticides

The insecticides

- **Dynamic analysis**: Requires running the code or binary
- **Static Analysis (*)**: No need to run the application, though in many cases, your code must build

Dynamic analysis tools I love

Fuzzing

— Helps us find bugs in difficult to reach execution paths

— Generates random program input for a given target function

— Uses an initial corpus or set of input as seed data to create input values

— Complex parsing logic is usually a good use case

Fuzzing Tools

- **Go-Fuzz**: coverage-guided fuzzing
- **Gofuzz**: provides many helper functions and type bindings
- **/trailofbits/go-fuzz-utils**: Interface to produce random values for various data types and can recursively populate complex structures from raw fuzz data generated by go-fuzz
- **package fuzz**: Native fuzzing package (available in Go 1.18 and above)

Fuzzing is easier than you might think

```
package <package name>

import (
    "testing"
)

func FuzzWhataver(f *testing.F) {
    // Use f.Add to provide a seed corpus
    testcases := <struct with sample input>
    for _, tc := range testcases {
        f.Add(tc)
    }
    f.Fuzz(func(t *testing.T, <params>) {
        // fuzz it
    })
}
```

Go-Fuzz fuzzer

```go
func FuzzParseRequest(data []byte) int {
   type FuzzStructure struct {
      name string
      zone string
   }

   var testStructure FuzzStructure

   tp, err := go_fuzz_utils.NewTypeProvider(data)
   if err != nil {
      return 0
   }

   err = tp.Fill(&testStructure)
   if err != nil {
      return 0
   }

   parseRequest(testStructure.name, testStructure.zone)

   return 0
}
```

Native fuzzer

```go
func FuzzRequest(f *testing.F) {
	testcases := []struct {
		name, zone string
	}{
		{"inter.webs.test", "interwebs"},
		{"svc.inter.webs.pod", "interwebs"},
	}
	for _, tc := range testcases {
		f.Add(tc.name, tc.zone)
	}
	f.Fuzz(func(t *testing.T, name string, zone string) {
		t.Logf("Testing with name %s", name)
		parseRequest(name, zone)
	})
}
```

Fault Injection

- We use `krf`[8] to inject faults in the SUT
- Intercepts syscalls, io operations, and injects faults
- Often useful to reveal error handling issues in Go

[8] github.com/trailofbits/krf

Other Tools

- Gopter: Property testing
- package `testing`: Native unit testing package
- Go race detector: useful for catching some basic data races dynamically.
- go-deadlock: Helps catch deadlocks dynamically

Static Analysis: Welcome to the program analysis world

Program Analysis

- Programs that reason about programs
- Infer facts about a program by inspecting its code
- Without execution, for the most part (see Symbolic Execution)

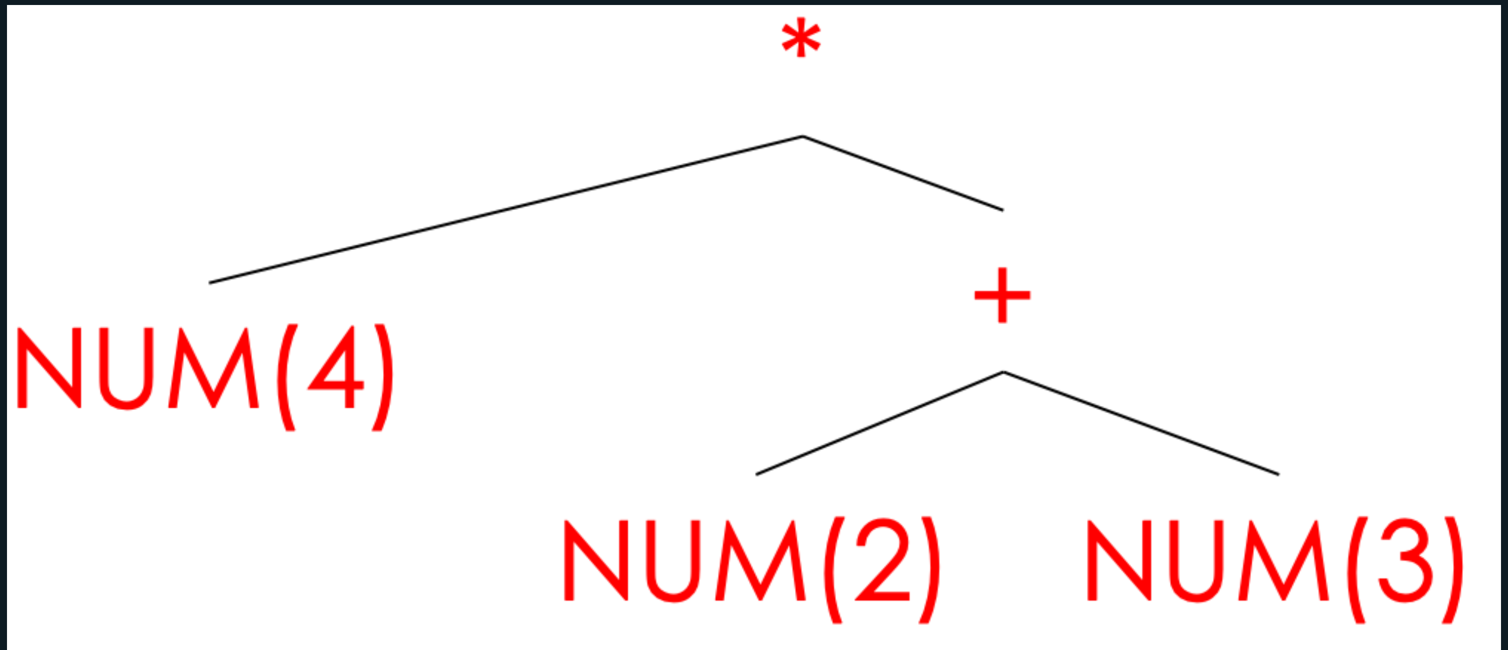
General Approach

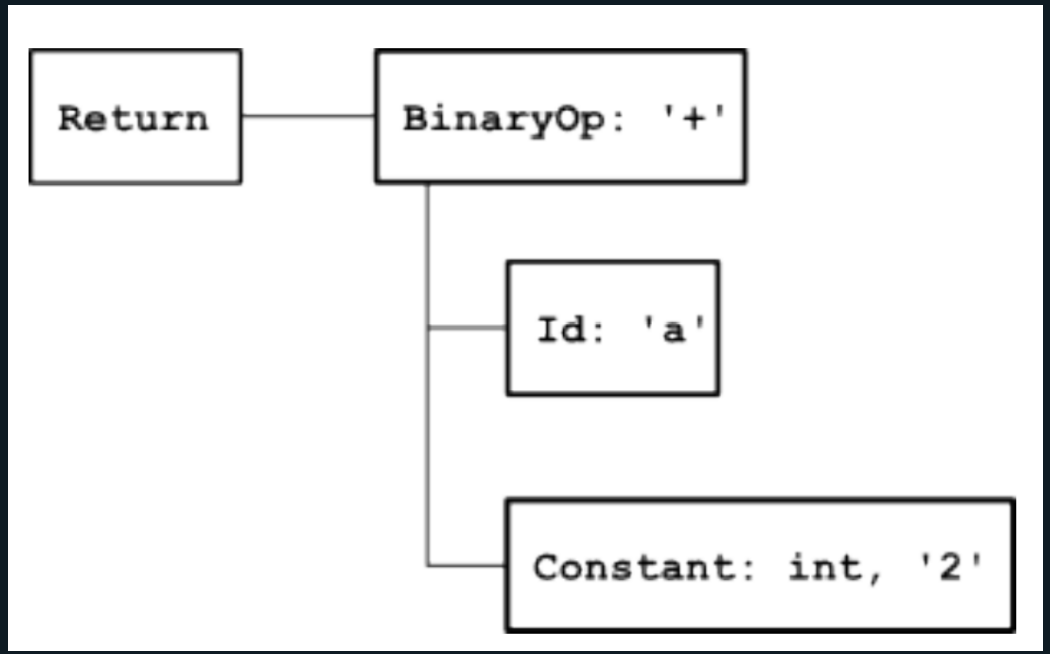
1. Generate an abstract interpretation of code
2. Remove the stuff that is not important for your analysis
   a. Build an Abstract Syntax Tree
   b. Build Control Flow Graph for data flow analysis
3. Conduct analysis on abstracted program

AST

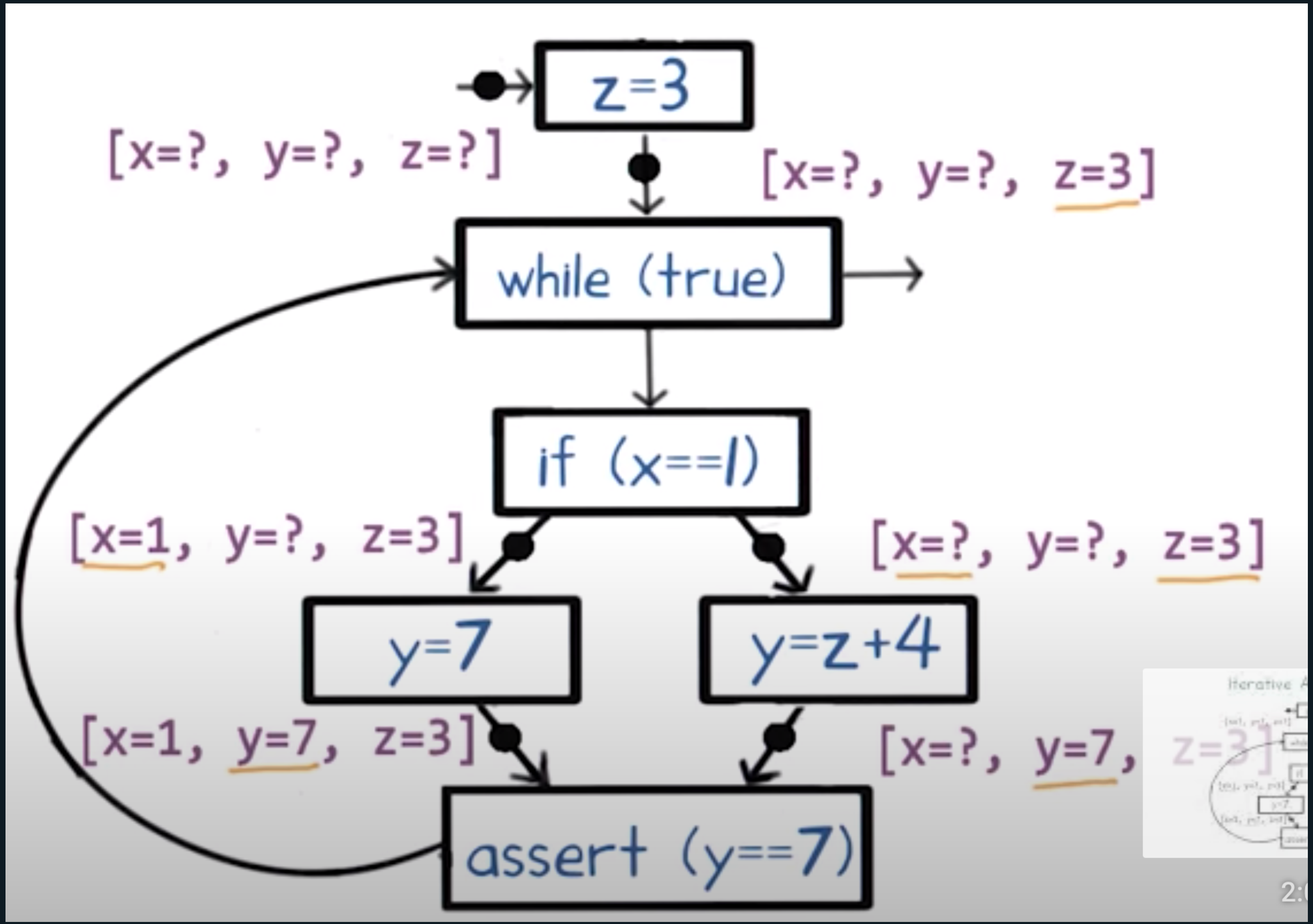
- Simplified syntactic representation of code
- Discards grammar not needed for analysis (i.e. `while`'s and`for`'s are all loops)

`4*(2+3) ------------------------- return a+2`





CFG


z=3
[x=?, y=?, z=?]
[x=?, y=?, z=3]
while (true)
if (x==1)
[x=1, y=?, z=3]
[x=?, y=?, z=3]
y=7
y=z+4
[x=1, y=7, z=3]
[x=?, y=7, z=3]
assert (y==7)

AST based tooling

- gosec
- errcheck
- ineffassign
- semgrep (*)
- ruleguard
- Go analyzers
- `go vet`

Gosec

Available rules

- G101: Look for hard coded credentials
- G102: Bind to all interfaces
- G103: Audit the use of unsafe block
- G104: Audit errors not checked
- G106: Audit the use of ssh.InsecureIgnoreHostKey
- G107: Url provided to HTTP request as taint input
- G108: Profiling endpoint automatically exposed on /debug/pprof
- G109: Potential Integer overflow made by strconv.Atoi result conversion to int16/3
- G110: Potential DoS vulnerability via decompression bomb
- G201: SQL query construction using format string
- G202: SQL query construction using string concatenation
- G203: Use of unescaped data in HTML templates
- G204: Audit use of command execution
- G301: Poor file permissions used when creating a directory
- G302: Poor file permissions used with chmod
- G303: Creating tempfile using a predictable path
- G304: File path provided as taint input
- G305: File traversal when extracting zip/tar archive
- G306: Poor file permissions used when writing to a new file
- G307: Deferring a method which returns an error
- G401: Detect the usage of DES, RC4, MD5 or SHA1
- G402: Look for bad TLS connection settings
- G403: Ensure minimum RSA key length of 2048 bits
- G404: Insecure random number source (rand)
- G501: Import blocklist: crypto/md5
- G502: Import blocklist: crypto/des
- G503: Import blocklist: crypto/rc4
- G504: Import blocklist: net/http/cgi
- G505: Import blocklist: crypto/sha1
- G601: Implicit memory aliasing of items from a range statement

Evaluating AST based tooling

— Simple issues are easy to catch fast with AST parsing
— Insecure usage of functions, configuration issues
— Lots of false positives
— Lack of data flow analysis
— Great when intraprocedural analysis is sufficient

AST analysis with Go

- `package go/ast`

- golang.org/x/tools/go/ssa

SSA?

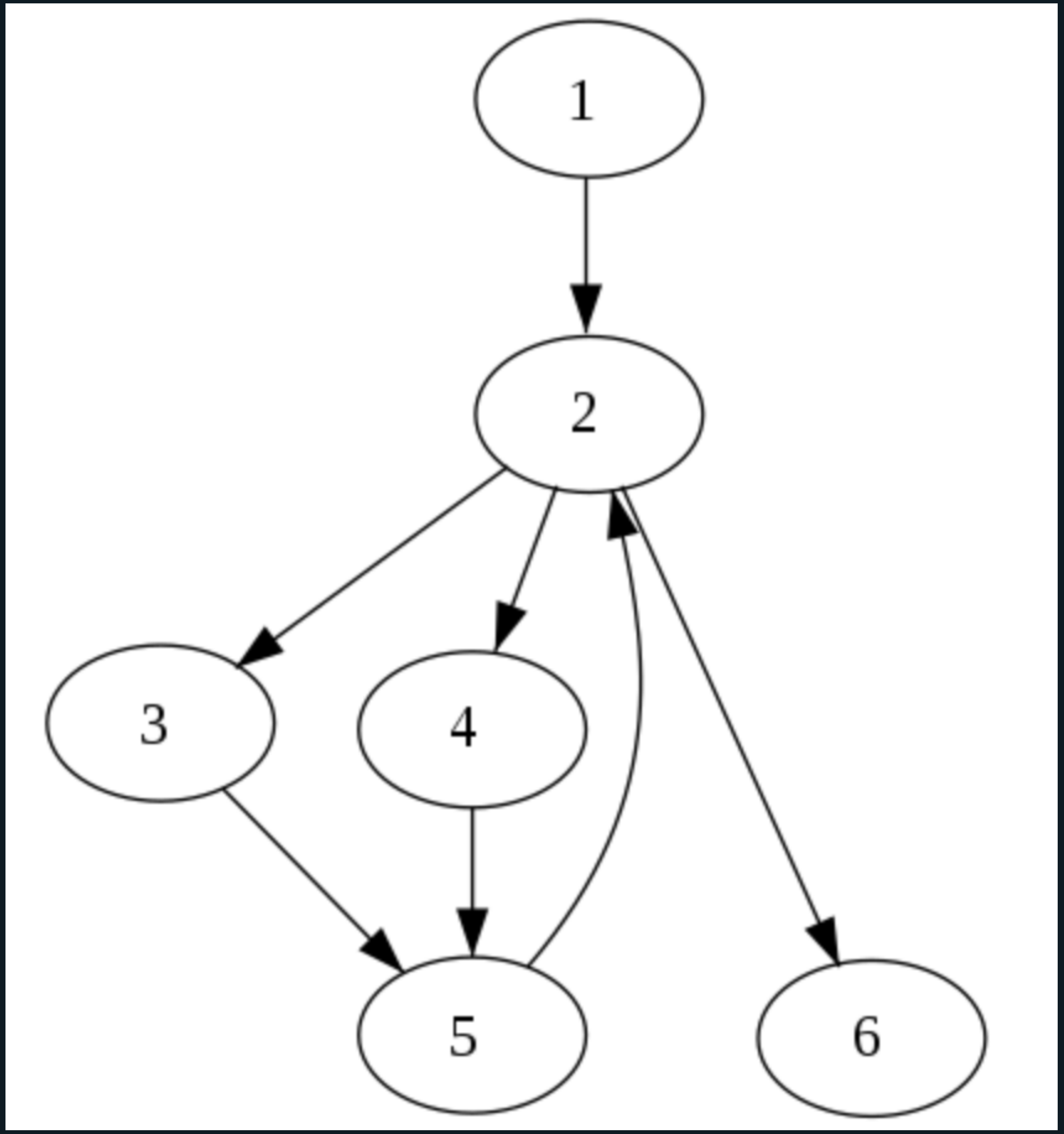
- static single-assignment
- IR representation used by Go
- Used by compiler to optimize code before generating Go assembly, conduct alias analysis, etc.
- builds a naive SSA form of the code
- determine dominator tree for each call
- we can now figure out execution paths by building call graphs
- allows us to answer questions like, "is parameter `foo' user-controlled?

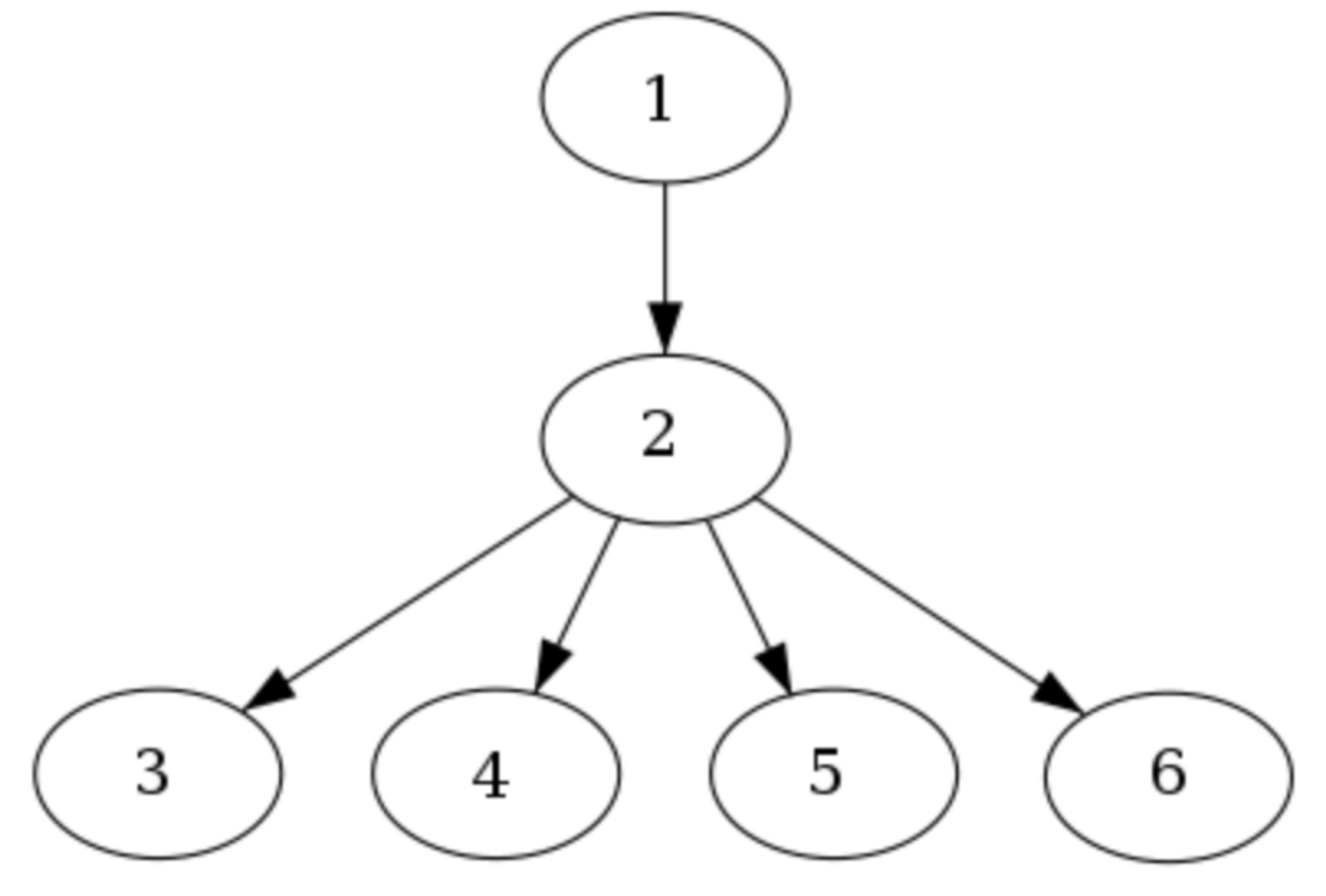
SSA Output

```go
package main
import "fmt"
const message = "Hello, World!"
func main() {
    fmt.Println(message)
}
```

```
func main():
0:                                                          entry P:0 S:0
    t0 = new [1]interface{} (varargs)                   *[1]interface{}
    t1 = &t0[0:int]                                        *interface{}
    t2 = make interface{} <- string ("Hello, World!":string)  interface{}
    *t1 = t2
    t3 = slice t0[:]                                      []interface{}
    t4 = fmt.Println(t3...)                          (n int, err error)
    return
```

Dominator Trees





SSA based tools

- github/codeql-go
- system-pclub/GCatch
- stripe-archive/safesql
- go-kart

CodeQL

- Interprocedural analysis
- Taint tracking
- Steep learning curve
- Slow

GCatch

- Catches concurrency bugs using SSA analysis
- Models Go structures for concurrency like channels
- Leverages Z3 for constraint solving
- Finds:
    - Goroutine leaks (misuse of channels)
    - Missing unlocks
    - Deadlock
    - Inconsistent field protections
    - Other concurrency bugs

GCatch in action

```
----------Bug[1]----------
        Type: BMOC      Reason: One or multiple channel operation is blocked.
-----Blocking at:
        File: /home/vagrant/go/src/github.com/[redacted].go:80
-----Blocking Path NO. 0
Call :/home/vagrant/go/src/github.com/[redacted].go:74:21       '✓'
Call :/home/vagrant/go/src/github.com/[redacted].go:75:20       '✓'
ChanMake :/home/vagrant/go/src/github.com/[redacted].go:78:17   '✓'
Chan_op :/home/vagrant/go/src/github.com/[redacted].go:80:2     Blocking
Call :/home/vagrant/go/src/github.com/[redacted].go:72:2        '✗'
End :/home/vagrant/go/src/github.com/[redacted].go:72:2         '✗'
```

Semgrep (*) [9]

- Intraprocedural analysis
- More than grep
- Uses generated tree-sitter grammars
- Can catch easy instances of issues found by GCatch (but faster)
- Missed 3/48 of BMOC bugs reported by GCatch to Moby/Docker
- 2/3 of missed bugs require interprocedural analysis

[9] trailofbits/semgrep-rules

Semgrep [10]

```yaml
rules:
- id: missing-unlock-before-return
  patterns:
    - pattern-either:
      - pattern: panic(...)
      - pattern: return ...
    - pattern-inside: |
        $T.Lock()
        ...
        $T.Unlock()
    - pattern-not-inside: |
        $T.Unlock()
        ...
    - pattern-not-inside: |
        defer $T.Unlock()
        ...
  message: |
    Missing mutex unlock before returning from a function.
    This could result in panics resulting from double lock operations
  languages: [go]
  severity: ERROR
```

[10] Semgrep Playground

How about misuse of non-native packages?

misuse of non-native packages: tooling

— We can use semgrep for a lot of use cases

— CodeQL for more in-depth analysis (slow)

Insecure CSRF key with Gorilla

```go
import (
    "encoding/json"
    "net/http"

    "github.com/gorilla/csrf"
    "github.com/gorilla/mux"
)

var (
    key = []byte("insecurekey")
)

func main() {
    r := mux.NewRouter()

    csrfMiddleware := csrf.Protect(key)

    api := r.PathPrefix("/api").Subrouter()
    api.Use(csrfMiddleware)
    api.HandleFunc("/user/{id}", GetUser).Methods("GET")

    http.ListenAndServe(":8000", r)
}
```

Overly Permissive CORS with Gorilla

```go
import (
    "net/http"
    "io"
    "github.com/gorilla/handlers"
)

var (
    tooPermissive  = []string{"*"}
)
func main () {
    serveMux := http.NewServeMux()

    originHeaders := handlers.AllowedOrigins(tooPermissive)

    serveMux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        io.WriteString(w, "Hello World!")
    })

    http.ListenAndServe(":8080", handlers.CORS(originHeaders)(serveMux))
}
```

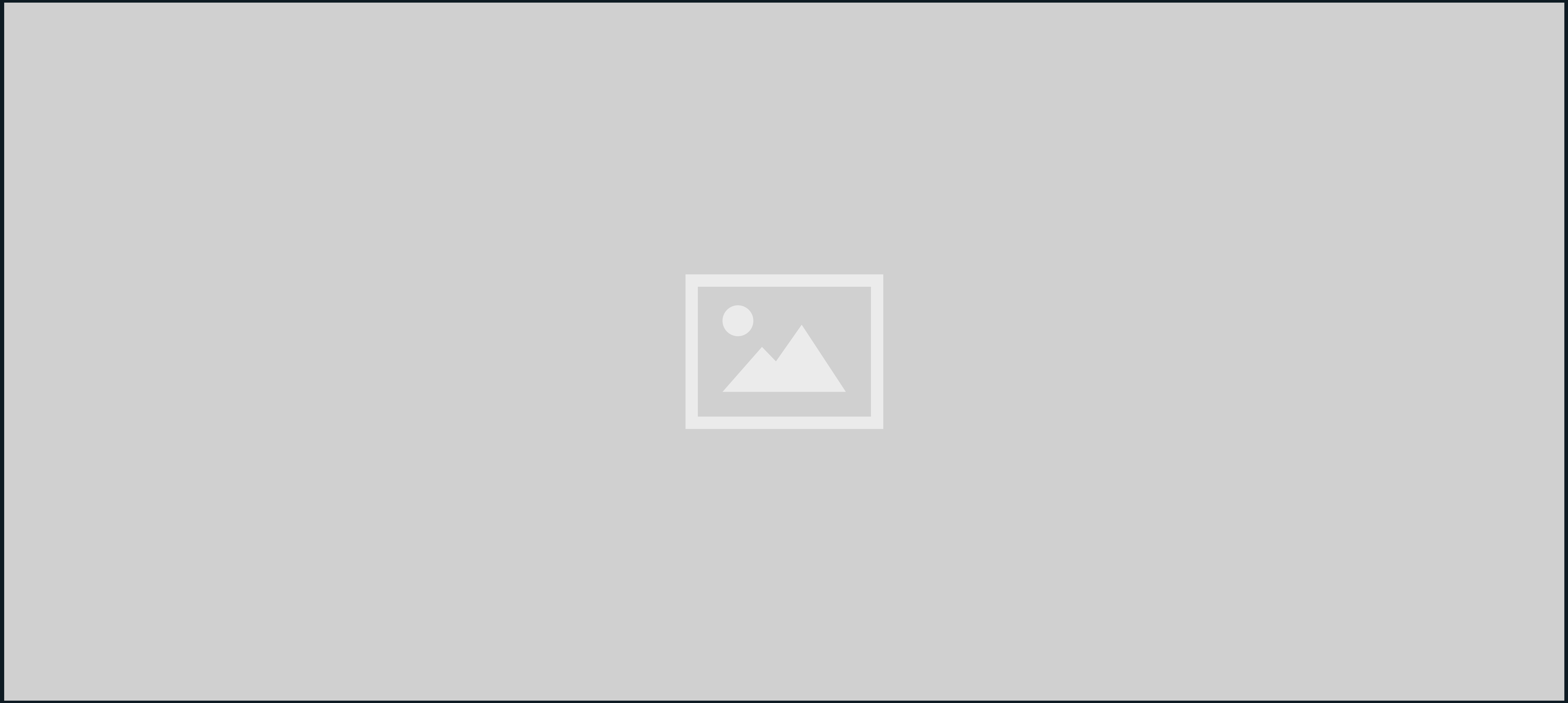
Insecure use of Gorm

```
db.Exec("INSERT INTO people (name) VALUES(\""+uname+"\")")
// (...)
db.Raw("SELECT id, name, age FROM users WHERE name = ?", userName)
```

Gotico

- In-development
- Analyzers for
  - Gin
  - Gorilla
  - Tendermint
  - Gorm
- Relies heavily on AST parsing
- Basic SSA support for answering generic tain analysis questions

Gotico



Why not just stick to semgrep and CodeQL

— A custom solution allows us to control the logic for each bug

— This is often needed given how different packages are used

— **Goal**: make it as easy as possible to write new rules

— **Goal**: so easy it encourages others to contribute new rules

Wrap up

- Leverage dynamic analysis as much as possible
  - Review unit tests
  - Fuzz complex logic
  - Test edge cases with `krf`
- Use AST tools for finding common misuse of native packages, interprocedural analysis
- Use SSA based tools for complex bugs that require intraprocedural analysis reduced FPs
- Use Semgrep, CodeQL, ruleguard when custom rules are needed (and finding instances of a specific issue)

Build your own

— Or contribute to existing tools

Resources

- trailofbits/not-going-anywhere
- amit-davidson/GopherCon2021IsraelStaticAnalysisWorkshop
- Introduction to Automated Analysis
- Software Analysis & Testing - Georgia Tech
- SSA Book
- SSA Playgroud

ToB Resources

- trailofbits/not-going-anywhere
- trailofbits/semgrep-rules
- trailofbits/krf
- trailofbits/publications