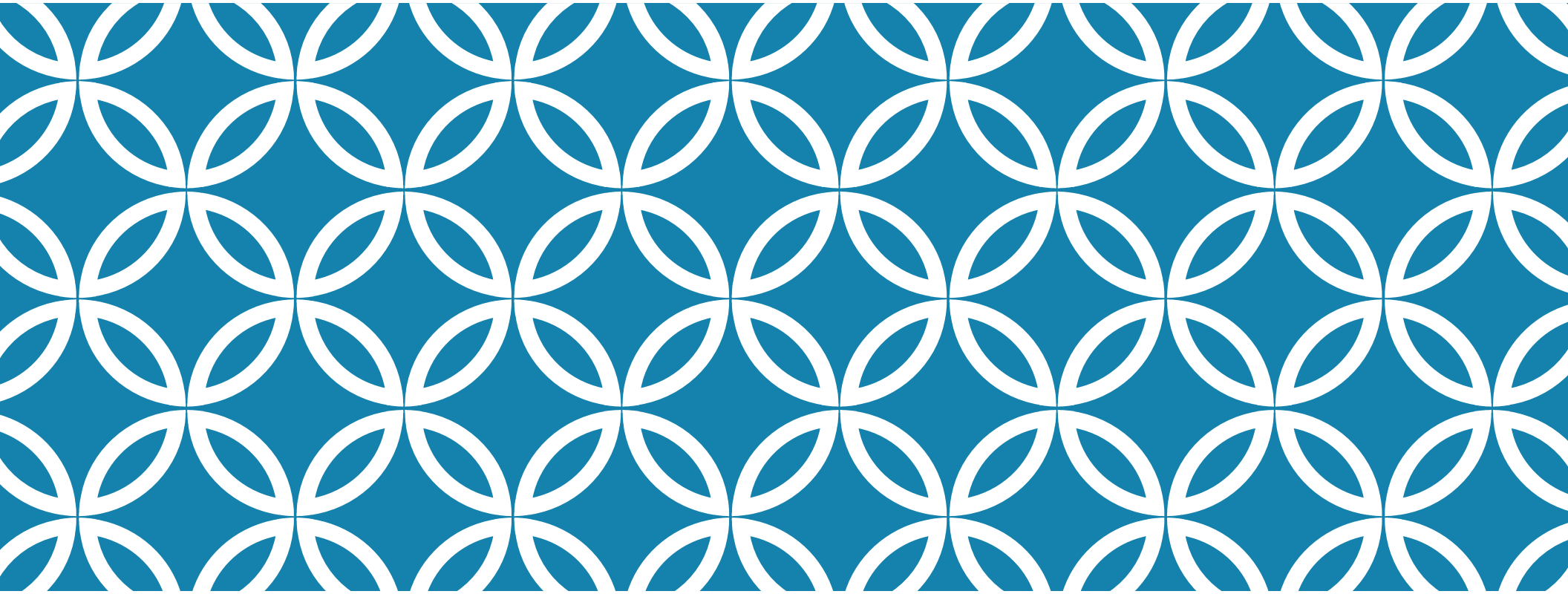


FUZZING AND PROPERTY-BASED TESTING IN GO

Mark Gritter

February 19, 2020

Minneapolis Golang Meetup

TL;DR

`go-fuzz`: coverage-guided exploration of a function

- large number of different byte inputs
- "Fuzzing"

`testing/quick` and `gopter`: randomized testing

- structured inputs
- checking an invariant
- "Property-based testing."

HOW I LEARNED TO START WORRYING

Talk: “Never Trust Any Published Algorithm”

Experiments:

- JSON parsing? (C)
- Replication protocol? (C++)
- Steem Blockchain protocol? (C++) --- and its JSON parser had other problems– so does Go’s.
- PKCS11 parser? (Go)
- KMIP implementation? (Go)

My estimate: *80% of code that’s never been fuzzed will exhibit bugs*

GO-FUZZ

Compiles an instrumented binary

Monitors “path” coverage (more on this later)

Generates inputs, focusing on variations of those that expanded coverage

Records inputs that cause a crash, and builds a corpus that exhibits good coverage

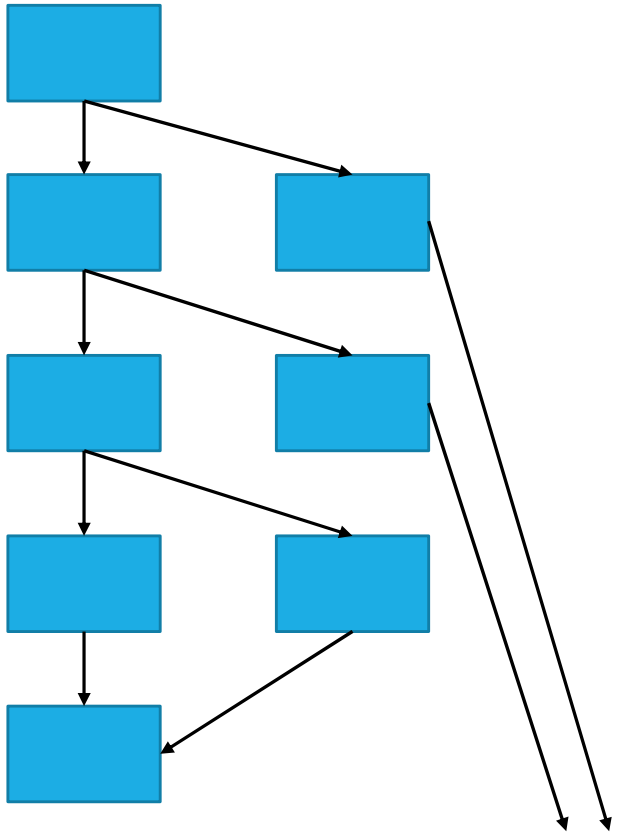
EXAMPLE: MPEG TRANSPORT STREAMS

https://github.com/jdeisenh/gots

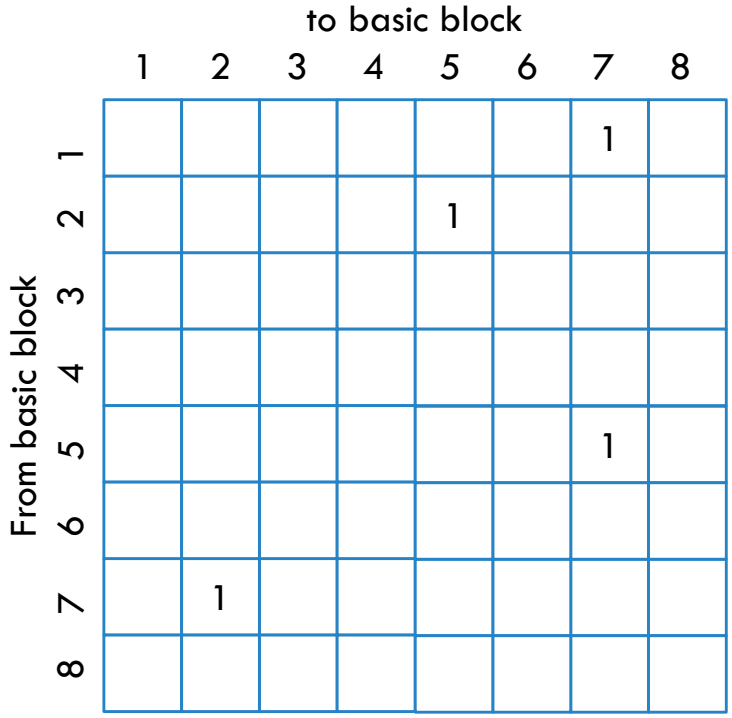
PATH COVERAGE – BASIC BLOCKS

```
func newPacket(data []byte)
(*Packet, error) {
     if len(data) != PacketSize {
          return nil, ...
     }
     if data[0] != SyncByte {
          return nil, ...
     }
     p := &Packet{}

     if p.ContainsAdaptationField {
          ...
```



PATH COVERAGE – MATRIX



Add a "1" to the matrix whenever a basic block transition occurs.

The full matrix would be huge so instead a fixed-sized bitmap is used, and matrix entries are hashed into bitmap bits.

Coverage = # of different coverage bitmaps

GENERIC FUZZING ADVICE

Disable checksums or signatures, or have the fuzzing harness fill them in correctly.

- An attacker can be assumed to compute a checksum correctly.

Small examples >> big examples

- Corpus doesn't need to be big, but it's useful to have examples that are small so tests run faster.

Fix known bugs and restart

- Fuzzing operates faster when it's not running into the same crashes over and over

Consider changing your randomized tests into fuzz tests

- Every time you would get a random number, read it from the fuzzing input.
- This doesn't really work well with the sort of mutations that fuzzers make, but tracking coverage is a big win.

TESTING/QUICK

Standard Go module

Used with testing module

Runs a function with random input and tries to get it to return false!

EXAMPLE: PERCENT-ENCODING

TESTING/QUICK AND YOUR OWN TYPES

Implement the Generator interface:

```
type Generator interface {
    // Generate returns a random instance of the
    // type on which it is a method using the size
    // as a size hint.
    Generate(rand *rand.Rand, size int) reflect.Value
}
```

PROPERTY IDEAS

Reference: Fast implementation = Slow implementation

Idempotence: F(F(x)) = F(x), for example “to uppercase” or “sort”

Encode/Decode: F(G(x)) = x = G(F(x))

Commutivity: F(G(x)) = G(F(x)), sometimes

Invariants: I(x) = I(F(x)), if you know one for the data structure

Induction: F(xy) = x # F(y), useful for tree structures or other decomposable problems

See the great article at

https://blog.ssanj.net/posts/2016-06-26-property-based-testing-patterns.html

GOPTER

Inspired by the original Haskell QuickCheck.

Not a “frozen” (dead) project, like testing/quick.

Explicit choice of input generators.

Composable generators.

Minimization of failing test cases!

EXAMPLE: PERCENT-ENCODING (AGAIN)

GOPTER AND YOUR OWN TYPES

```
type Gen func(*GenParameters) *GenResult

type GenParameters struct {
    MinSize         int
    MaxSize         int
    MaxShrinkCount int
    Rng             *rand.Rand
}

func NewGenResult(result interface{}, shrinker Shrinker)
*GenResult
```

“ARBITRARIES”

Use reflection to select the generator instead, just give it a function (like testing/quick):

```
func (a *Arbitraries) ForAll(condition interface{}) gopter.Prop
```

You can still register a generator for unknown types, or override the defaults:

```
func (a *Arbitraries) RegisterGen(gen gopter.Gen)
```

COMPOSING GENERATORS

```
gen.Struct( reflect.Type, map[string]Gen ) Gen

gen.MapOf( keygen, elementGen Gen ) Gen

gen.SliceOf( elementGen Gen, ...reflectType ) Gen
```

These aren't very well documented (it would be nice to have more examples.) But you can create a data structure by specifying generators for its individual elements!

STRENGTHS AND WEAKNESSES

A software correctness tool has value because it has a *different* set of biases than human programmers and testers.

- Methodical, but dumb.
- Humans come in with an understanding of how code is "supposed to work"; tools do not.
- Humans are great at experimenting and exploring… for about 20 tests, then they get bored.

Use these tools when you have a lot of cases to worry about, and can succinctly describe what is "correct".

THANKS FOR ATTENDING!

Mark Gritter
Vault Advisor at HashiCorp (we're hiring!)
Twitter: @markgritter
Github: mgritter

go-fuzz: https://github.com/dvyukov/go-fuzz

testing/quick: https://golang.org/pkg/testing/quick/

gopter: https://github.com/leanovate/gopter

Other tools to check out:

- **Hypothesis**: property-based testing in Python)
- **American Fuzzy Lop**: fuzzer for C, C++, Objective C, or block-box binaries (with QEMU on Linux)

Come see me at MinneBar: **Graph Grammars -- and Failure in Language Design**