Fuzzing in Go

Stack-Based VM Example

Moritz Gartner

12.12.2024

Project

Virtual Machine

Stack of Tokens[]

ONE
TWO
ONE
PLUS
MULT

eval(): float
convert(): Expression

Expression

AST representation

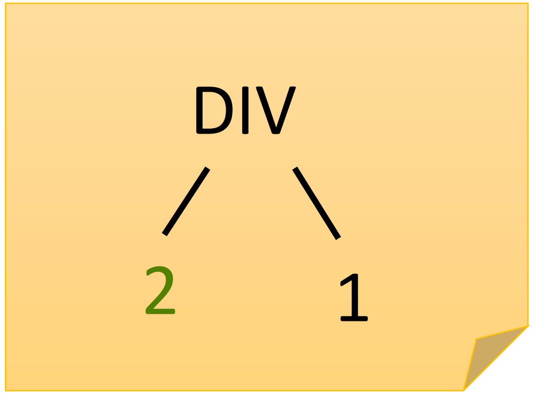
```
      *
    /   \
   1     +
        / \
       2   1
```

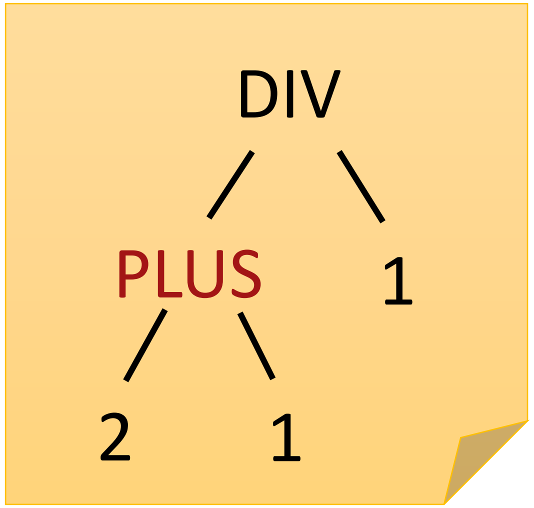
eval(): float
convert(): Token[]

Bug

```go
func (exp DivExp) Eval() float64 {
    // == BUG
    switch exp.Left.(type) {
    case *IntExp:
        // do nothing
    default:
        fmt.Println("Bug hit. Left exp is: ", exp.Left)
        return 0
    }
    // ==

    return exp.Right.Eval() / exp.Left.Eval()
}
```





Test Case

```go
func TestDivBug(t *testing.T) {
	// arrange
	exp := stackvm.NewDivExp( //
		stackvm.NewPlusExp( //
			stackvm.NewIntExp(2), stackvm.NewIntExp(1)), //
		stackvm.NewIntExp(1))

	// act
	vmCode := exp.Convert()
	vm := stackvm.NewVM(vmCode)
	resultFromExp := exp.Eval()
	resultFromVM := vm.Run()

	// assert that Exp.eval == VM.run
	if resultFromExp != resultFromVM {
		t.Log(stackvm.Show(vmCode))
		t.Logf("Result from VM: %g", resultFromVM)
		t.Logf("Result from Expression: %g", resultFromExp)
		t.Errorf("Mismatch: Exp yields %g, VM yields %g", resultFromExp, resultFromVM)
	}
}
```

Test Case

```
=== RUN   TestDivBug
Bug hit. Left exp is:  &{0xc0001043b8 0xc0001043c0}
    c:\Users\moritz.gartner\repos studium\Fuzzing-in-Go\impl\stackvm\stack_vm_test.go:64: 1 2 1 + /
    c:\Users\moritz.gartner\repos studium\Fuzzing-in-Go\impl\stackvm\stack_vm_test.go:65: VM yields: 0.3333333333333333
    c:\Users\moritz.gartner\repos studium\Fuzzing-in-Go\impl\stackvm\stack_vm_test.go:66: Exp yields: 0
    c:\Users\moritz.gartner\repos studium\Fuzzing-in-Go\impl\stackvm\stack_vm_test.go:67: Mismatch, delta is 0.3333333333333333
--- FAIL: TestDivBug (0.00s)
FAIL
FAIL	project/impl/stackvm	0.360s
```

Generators

```go
func FuzzWithGenerator(f *testing.F) {
	// No need to add seed inputs as the generator will generate random expressions

	f.Fuzz(func(t *testing.T, seed int) {
		// use seed for reproducibility
		rand := rand.New(rand.NewSource(int64(seed)))

		// use generator to create a random expression
		exp := gen.RandomExp(rand, 3)

		// ... run the test case
	})
}
```

Guided Fuzzing

```go
func FuzzPlusExp(f *testing.F) {
	ff := fuzzplus.NewFuzzPlus(f)
	rand := rand.New(rand.NewSource(1))

	// create a test corpus of 500 random expressions
	for i := 0; i < 500; i++ {
		exp := gen.RandomExp(rand, 2)
		encodedExp, err := encoding.EncodeWithDepth(exp, 3, 0) // make data compatible with the fuzzer
		if err != nil {
			f.Fatalf("Failed to encode expression: %v", err)
		}
		ff.Add2(encodedExp)
	}

	ff.Fuzz(func(t *testing.T, in []encoding.EncodedExp) {
		expression, err := encoding.Decode(in, 3) // convert back into our data structure
		if err != nil {
			return
		}
		// ... run the test case
	})
}
```

Comparison

	Generators	Guided Fuzzing
Bug found after	0.27s	2.55s
Expression	1 2 + 1 1 * / 2	1 1 1 + /
Pros	• Easy to implement • Very fast • All inputs are valid	• May find more edge-cases • Tries to minimize the failing input
Cons	• Only as good as randomness • Failing input is likely to be large	• No native support for structs → cumbersome to implement → encoding/decoding adds new logic • Slower → performance depends on how many inputs come out valid after decoding

→ QuickCheck-style generators are the better choice

Further Literature

Last checked 11.12.2024:

- https://github.com/Lefted/Fuzzing-in-Go
- https://sulzmann.github.io/SoftwareProjekt/lec-cpp-advanced-vm.html
- https://hackage.haskell.org/package/QuickCheck-2.15.0.1/docs/Test-QuickCheck-Gen.html
- https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf
- https://sulzmann.github.io/SoftwareProjekt/lec-cpp-advanced-quick-check.html#(1)

Bonus

Generators 1/2

```go
func randomIntExp(rand *rand.Rand) stackvm.Exp {
	value := rand.Intn(2) + 1
	return stackvm.NewIntExp(value)
}

func randomPlusExp(rand *rand.Rand, depth int) stackvm.Exp {
	left := RandomExp(rand, depth-1)
	right := RandomExp(rand, depth-1)
	return stackvm.NewPlusExp(left, right)
}
```

Generators 2/2

```go
func RandomExp(rand *rand.Rand, depth int) stackvm.Exp {
	if depth <= 0 {
		return randomIntExp(rand)
	}

	operator := rand.Intn(4)
	switch operator {
	case 0:
		return randomIntExp(rand)
	case 1:
		return randomPlusExp(rand, depth)
	case 2:
		return randomMultExp(rand, depth)
	case 3:
		return randomDivExp(rand, depth)
	default:
		return randomIntExp(rand)
	}
}
```

Why not swap the operands of Div?

Here's why:

1. The guided fuzzer receives a seed of randomly generated expressions
2. It will then first try to find any bug using the provided seed inputs
3. After that it will try to mutate inputs to find new bugs

Swapping the operands:

The bug is likely to be found with the initial seed inputs
→ The guided fuzzer won't have anything to do

Our complex bug:

We can only provide randomly generated expressions with a depth of 1 as seed input
→ The seed input will not trigger the bug
→ The guided fuzzer will have to 'find out' that it can use a depth of 2 to find the bug