Can a Fuzzer Match a Human?

Solidity Case Study

Bhargava Shastry
Ethereum Foundation

@ibags
bshastry

Fuzzer No Match for Human Tester, but...

- It can find security-critical bugs that a tester may have missed
    - Often elicits: “Oh, I hadn’t considered that!”
- Throw the kitchen sink at something
- Really useful for differential (A/B) testing

tl;dr:

- Threat model: Incorrect code generation
- Randomly generated valid Solidity programs test compiler
- Found 22 bugs using semantic fuzzing
- Continuous fuzzing for early bug discovery
- Virtually no Yul optimizer bugs post release in two years

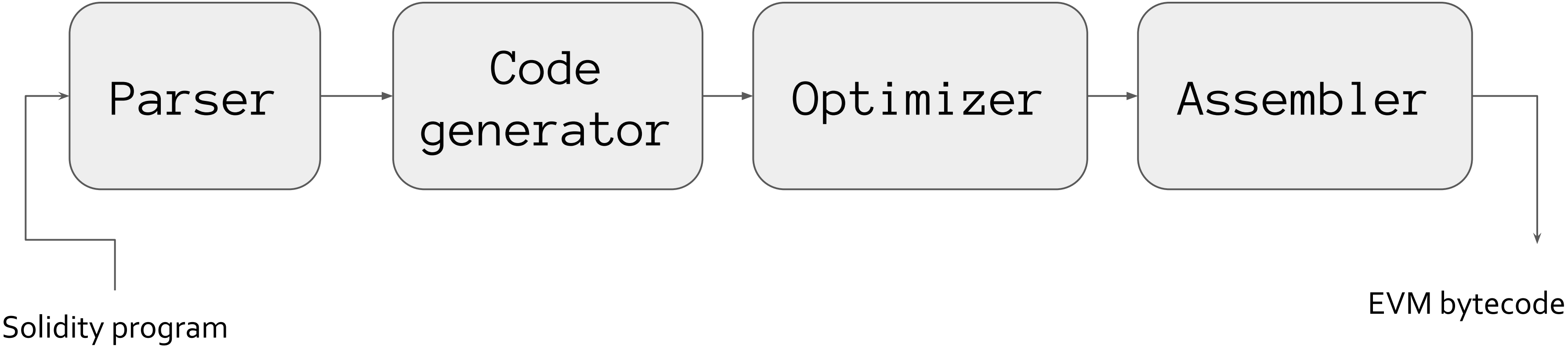
whoami

- Security engineer, Solidity team
- Semantic testing of Solidity compiler

Find security-critical bugs in the compiler before it is shipped

Introduction

Compiler Overview



Parser

Code generator

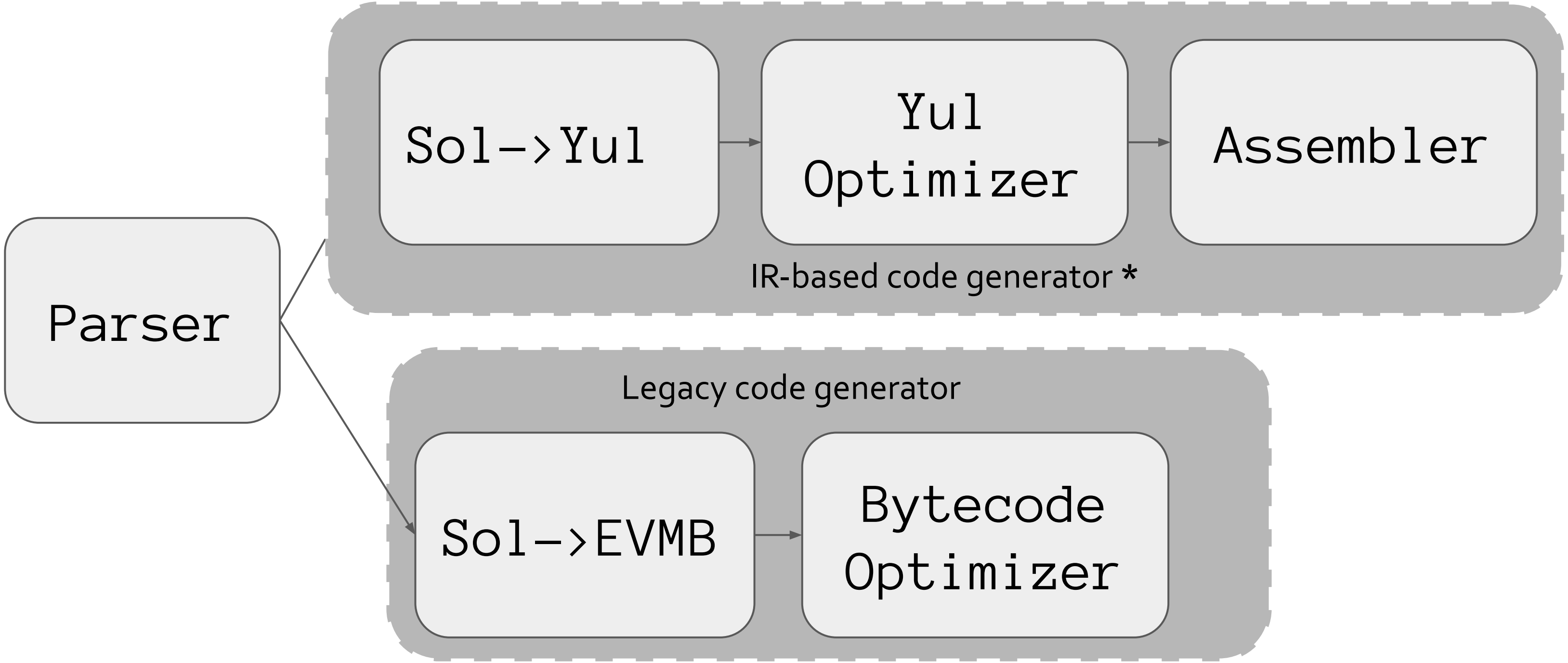
Optimizer

Assembler

Solidity program

EVM bytecode

Code generators



* IR-based code generator also makes use of bytecode optimizer

Threat model

- Compiler user (programmer) is not malicious
- Bugs introduced by the compiler itself
  - Optimizer(s)
  - Code generator(s)
  - Assembler
- Parser bugs are out of scope

Fuzz testing in a nutshell

```
while not ctrl + c
do
    input=gen_input()
    runProgram(input)
done
```

Limitation of random fuzzing

```
contract C {
   function foo()
public {

do_something();
   }
}
```

Mutation

```
contract C {
   fu#!3ion foo()
puX^&c {

do_something();
   }
}
```

Accepted by parser

Rejected by parser

Fuzzing a compiler requires generating valid programs…

… generating a valid program requires structure awareness

Approach

Input Generation

- Input generation approached in two different ways
    - Grammar-based Solidity program generator written in C++ only
    - Protobuf based Yul program generator written using protobuf C++ binding

Differential Testing

- Always compare two entities in order to find bug in one of them
  - Optimized and unoptimised
  - Legacy and IR based code generators
- Execution Tracing approached in two different ways
  - EVM client based
  - Yul interpreter based

Grammar based Input Generation

- A full-fledged Solidity program generator written in C++
- Each fuzzer mutation is a randomly-generator program
- All programs are semantically valid

Yul Input Generation

Specification written in protobuf language

```
message Block {
  repeated Statement stmts;
}
...
message program {
  repeated Block blocks;
}
```

Full spec:
https://github.com/ethereum/solidity/blob/develop/test/tools/ossfuzz/yulProto.proto

Input generation

- Input generated and mutated by libprotobuf-mutator
- Each input is a tree

```
blocks { stmts { ifstmt { condition {
binaryOp { eq { op1: varref{id: 0} op2: 0}
} } } } }
```

Input conversion

- Converter is source-to-source translator
- Input: protobuf serialization format
- Output: yul program

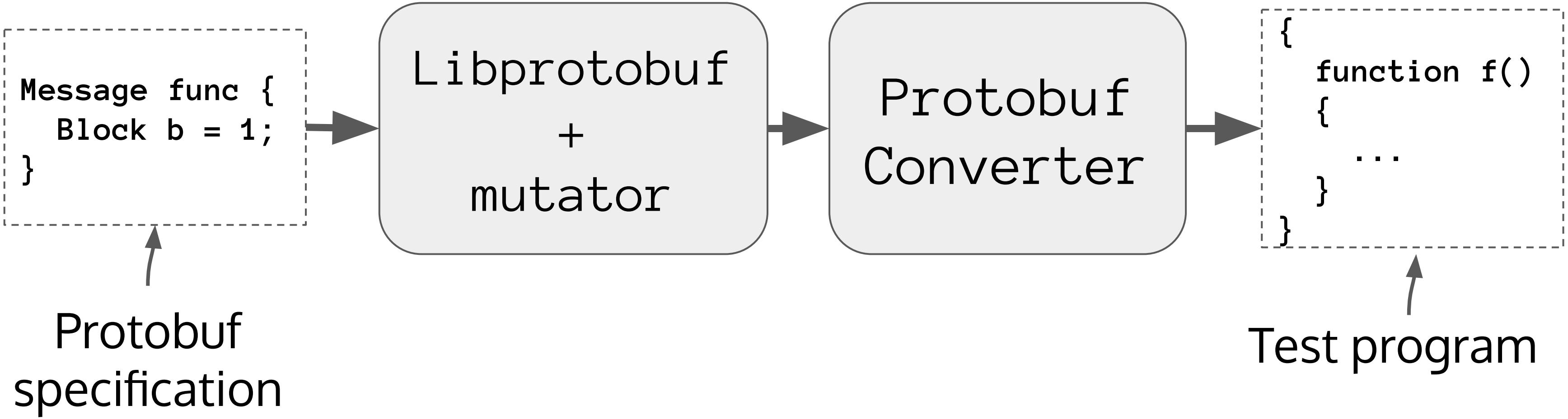
Example

```
blocks { stmts { ifstmt { condition {
binaryOp { eq { op1: varref{id: 0} op2: 0}
} } } } }
```

Conversion

```
if x_0 == 0
```

Test program generation



```
Message func {
  Block b = 1;
}
```

Protobuf specification

Libprotobuf + mutator

Protobuf Converter

```
{
  function f()
  {
    ...
  }
}
```

Test program

Correctness testing requires encoding expectation somehow

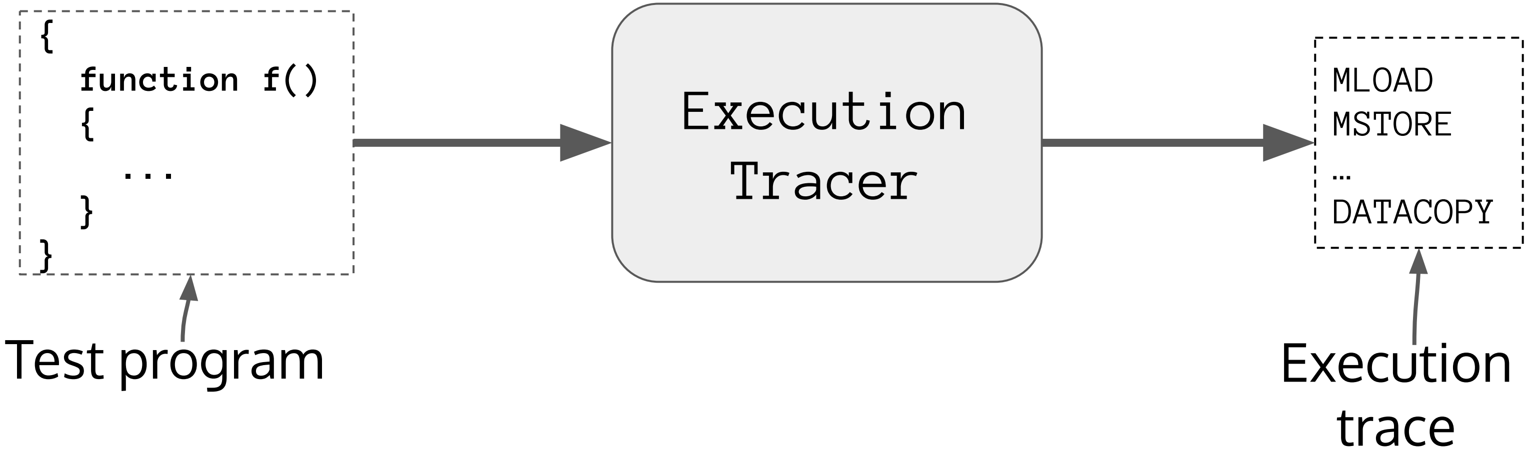
Differential testing

- Track side-effects of execution
- Run baseline and experiment programs
- Compare side-effects

Execution Tracing

- Solidity programs drive EVM client (Evmone)
- Yul programs drive the Yul interpreter

Execution Tracing Overview



```
{
  function f()
  {
    ...
  }
}
```

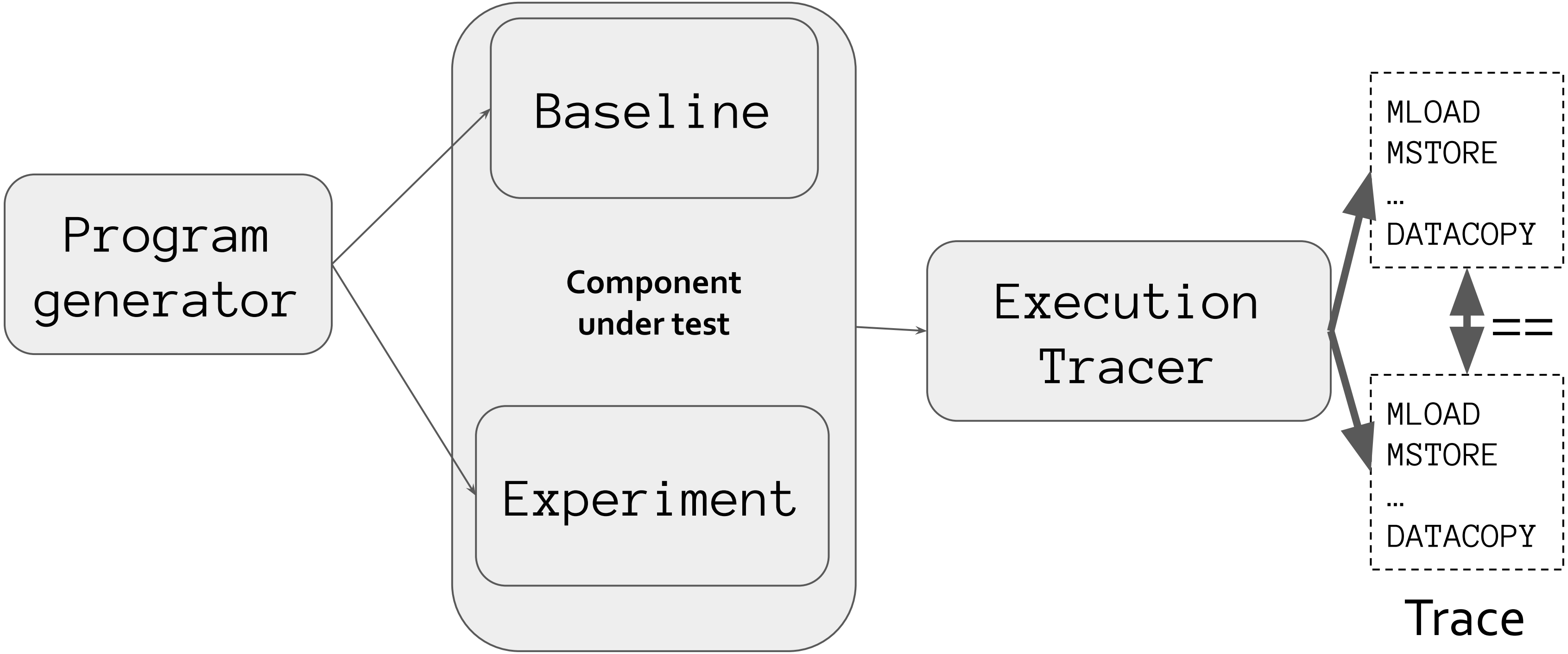
Test program

Execution Tracer

```
MLOAD
MSTORE
…
DATACOPY
```

Execution trace

Fuzzing Setup



Results

Bug 1: Incorrect keccak computation

```
contract C {
  function f() public returns (bool ret) {
    assembly {
      mstore(0, 0)
      let a := keccak256(0, 32)
      let b := keccak256(0, 23)
      ret := eq(a, b)
    }
  }
}
```

Compute keccak hash over memory contents addressed by [0, 31]

Compute keccak hash over memory contents addressed by [0, 23]

Are they equal?

Bug 1: Incorrect keccak computation

```
contract C {
  function f() public returns (bool ret) {
    assembly {
      mstore(0, 0)
      let a := keccak256(0, 32)
      let b := keccak256(0, 23)
      ret := eq(a, b)
    }
  }
}
```

Compute keccak hash over memory contents addressed by [0, 31]

Compute keccak hash over memory contents addressed by [0, 23]

Function returns true!

Bug 1: Root cause

- Compiler assumes keccak256 is computed over memory regions that are multiples of 32 bytes in size
- Caches based on start pointer
- Bug fix: Cache only if start pointer and length match

Bug 2: Incorrect Optimization

```
{
  function readValue() -> x {
    x := sload(0)
  }
  function writeValue() -> y {
    sstore(0, 2)
    y := sload(0)
  }
  function bug() -> z {
    z := mul(writeValue(), shl(readValue(), 1))
  }
}
```

Return value at storage zero

Write two to storage zero and
return two

??

Left-to-right evaluation

Correct Computation

```
function bug() -> z {
  z := mul(writeValue(), shl(readValue(), 1))
}
```

```
z := mul(2, shl(2, 1))
```

```
z := mul(2, 4):= 8
```

Optimizing Multiply by Two’s Power

$$X * 2^Y = X \ll Y$$

- 2^1 equivalent to left-shift by one
- Saves gas by eliminating multiplication

Incorrect Optimization

```
function bug() -> z {
  z := shl(readValue(), writeValue())
}
```

Arguments re-ordered

```
z := shl(0, 2)
```

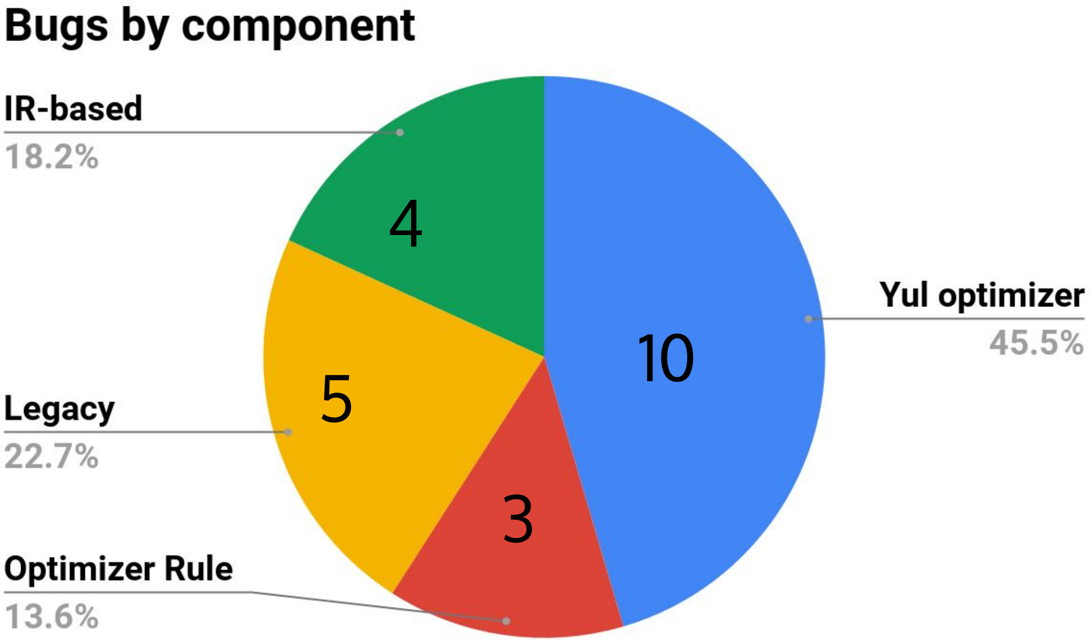
```
z := 2
```

Bug Fix: Incorrect optimization

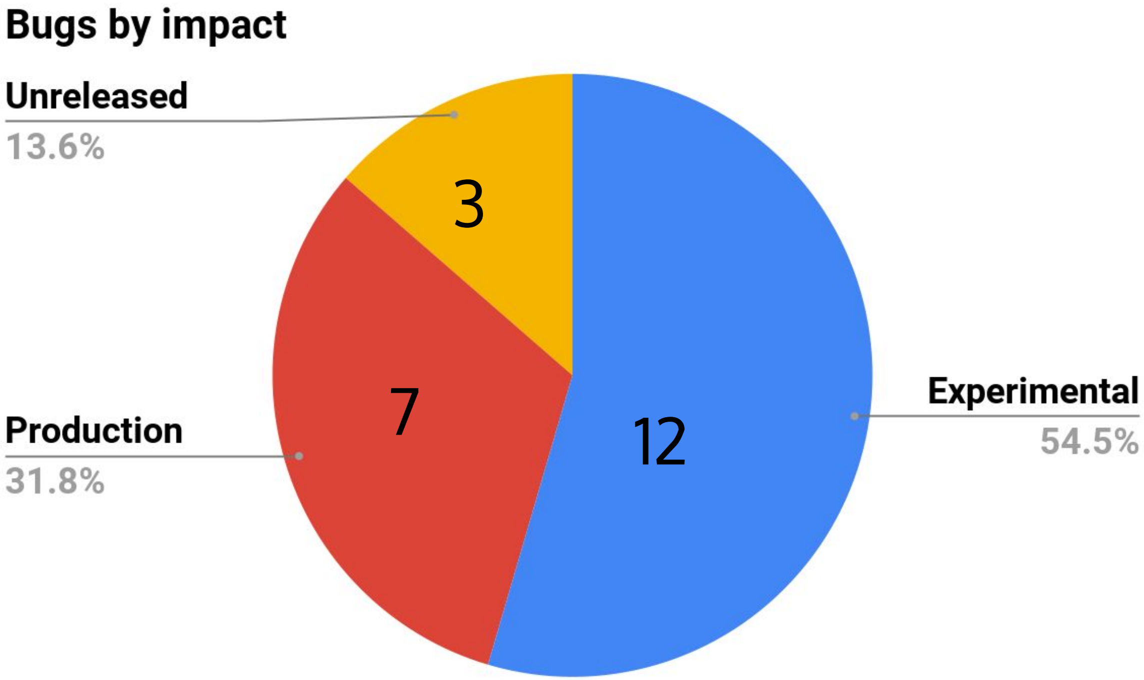
- Add safety check
- Optimization that impacts order of evaluation
    - Can only be applied if no side-effects
- The buggy test case would be unoptimized
    - But that’s a lot better than introducing a bug!

Summary

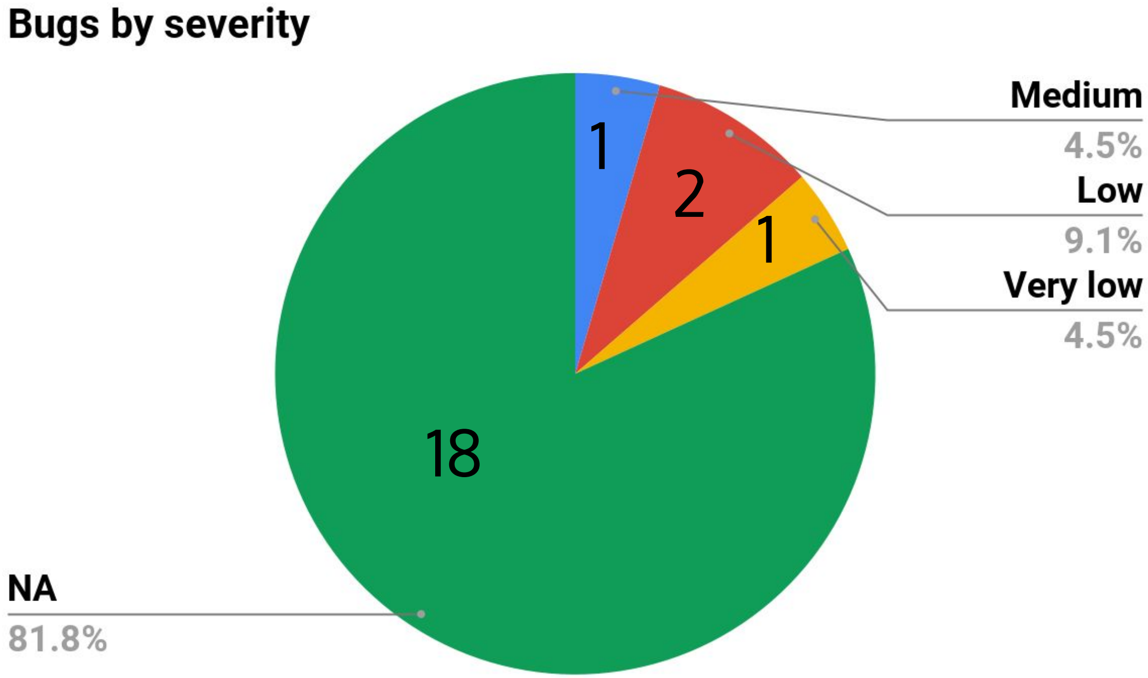
Bugs by component



Bugs by impact



Bugs by severity



Summary

- Three bugs found before PR merged!
- Fuzzing helped safely transition Yul optimizer from experimental to production
    - 15 bugs found before production release
- Zero bug collision with external users
    - Not present in real-world contracts?

Current Work

Two Bugs Required Human Assistance

- `returndatacopy(0, 1, 100)` inside a fallback function
  - HT @_hrkrshnn
- Storage corruption and empty push on bytes array
  - HT @ekpyron

Can Fuzzer Approach Humanness?

Heuristics + Randomness

- Pure randomness may be ill-suited sometimes
- Redundant memory store eliminator
    - Requires read location to be equal or not-equal to write location
    - Pure randomness will most likely not-equal than equal
    - Heuristic: Read from location that is already written to occasionally

Conclusion

Conclusion

- Continuous grammar-aware fuzzing for early bug discovery
- Useful for testing security-critical components of the Solidity Compiler
- Decent assurance
  - Evidence that it works
  - No formal guarantees though

Thank You!

ethereum/solidity.git

gitter.im/ethereum/solidity-dev

A tester's vain attempt to make their bug stand out in the next bug triage meeting

cartoontester.blogspot.com © 2013