

# Fuzzing the Solidity Compiler

Bhargava Shastry  
Ethereum Foundation



@bshastry



@ibags



bshastry



# whoami

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

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



## tl;dr:

- Threat model: Incorrect code generation
- Randomly generated **valid** Solidity (yul) programs test compiler
- Found **9 bugs** using semantic fuzzing
- **Continuous** fuzzing for early bug discovery



# Introduction

# Threat model

- Compiler user (developer) is not malicious
- Bugs introduced by the optimizer



# 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();  
    }  
}
```

Accepted by parser



Mutation

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

Rejected by parser

Fuzzing a compiler requires  
generating valid programs...

... generating a valid program requires  
structure awareness






# Approach

# Write a specification

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}  
} } } } }
```

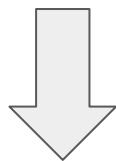
# Input conversion

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



# Example

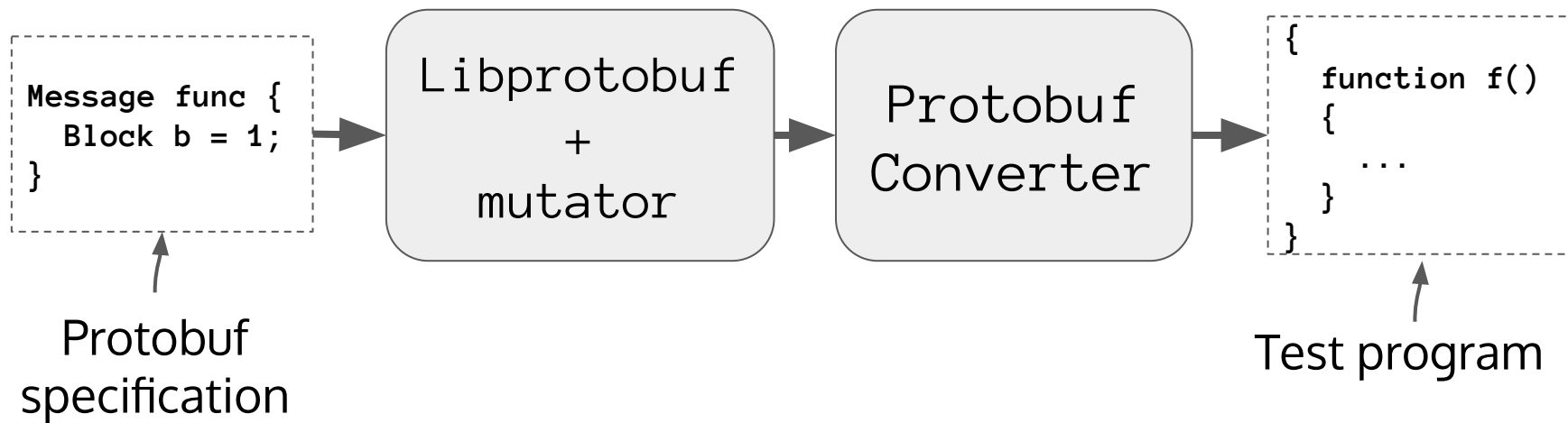
```
blocks { stmts { ifstmt { condition {  
binaryOp { eq { op1: varref{id: 0} op2: 0}  
} } } } }
```



Conversion

```
if x_0 == 0
```

# Test program generation



Correctness testing requires encoding expectation somehow

# Differential fuzzing

- Track side-effects of execution
- Run program
- Run optimized program
- Compare side-effects



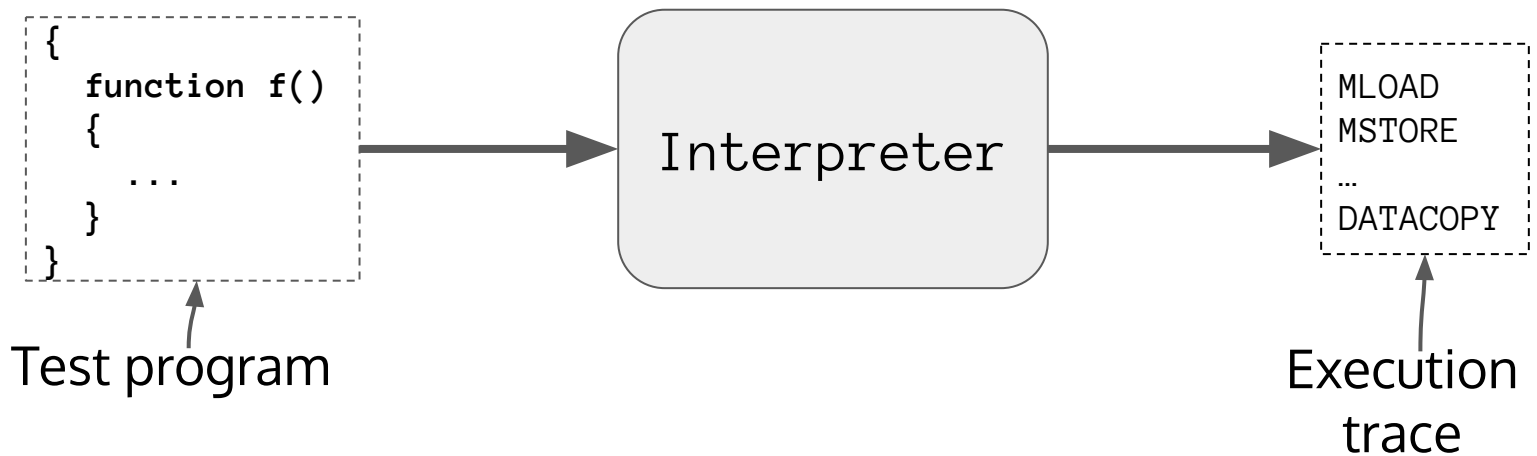


# Yul interpreter

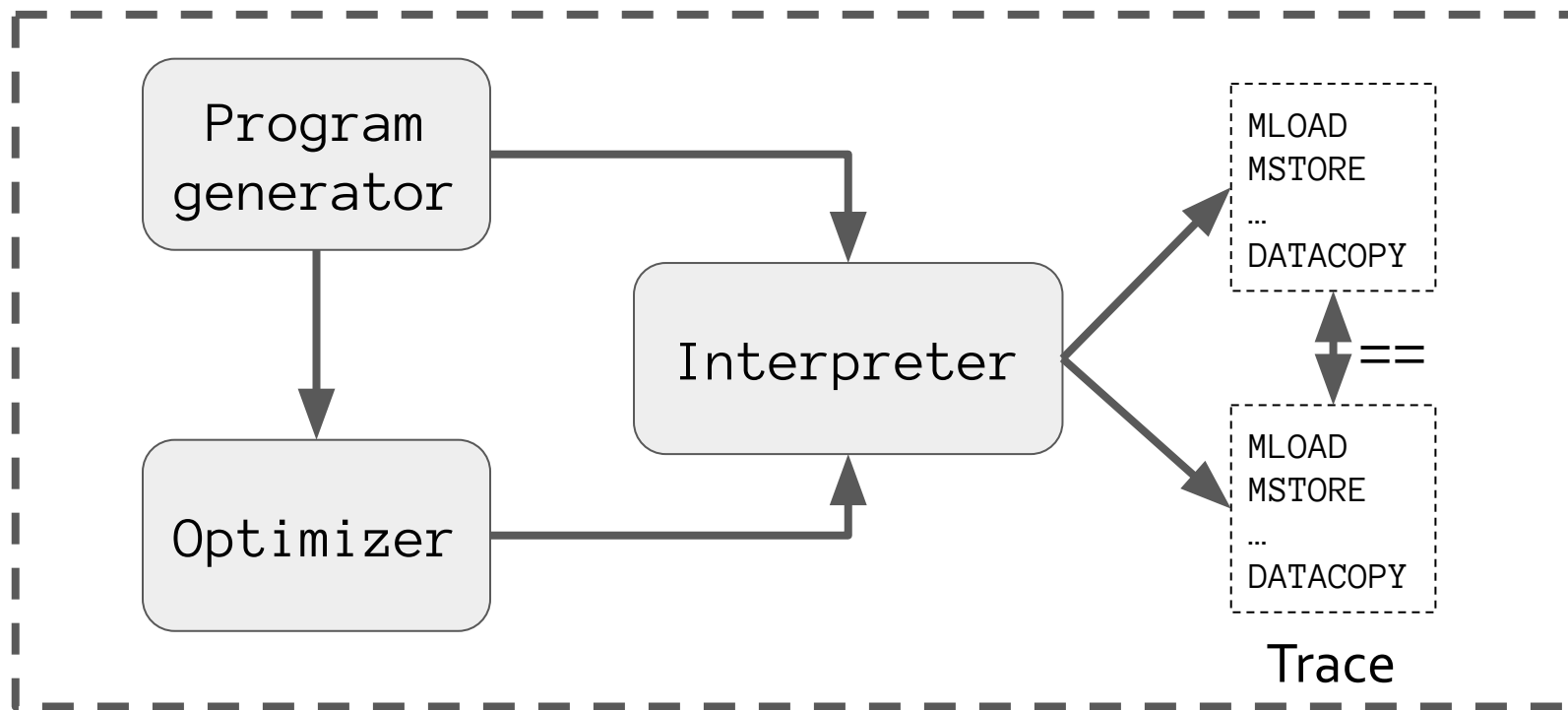
- Interprets arbitrary yul program
- Outputs side-effects as a trace (string)



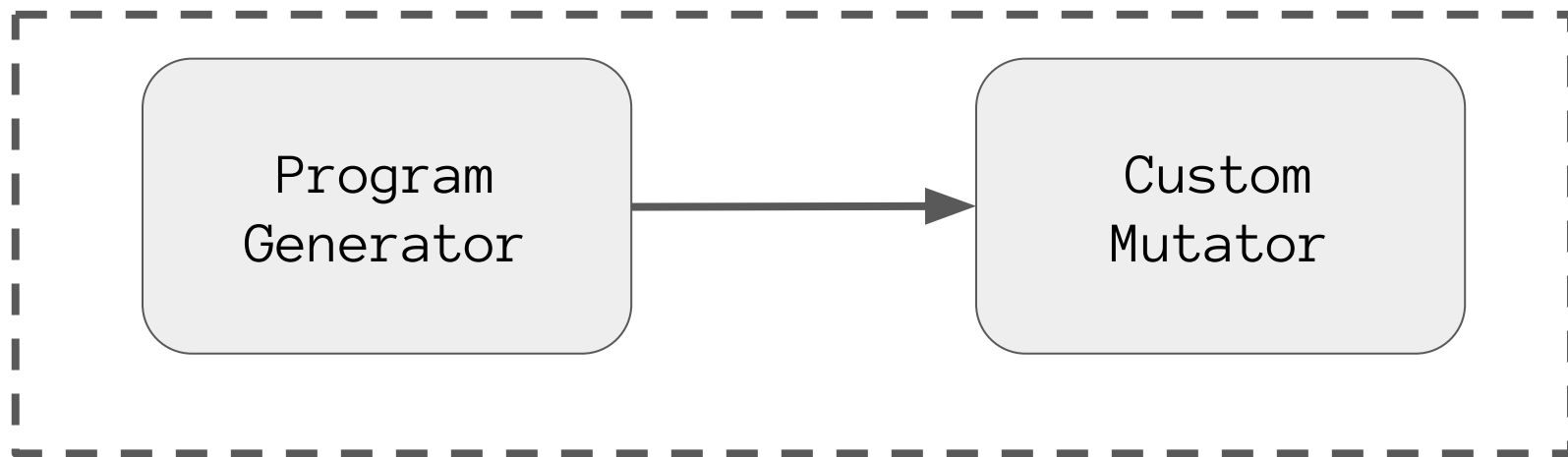
# Yul interpreter



# Fuzzing Setup



# Custom Fuzz Mutator



```
if x_0 == 0
```

```
if x_0 != 0
```

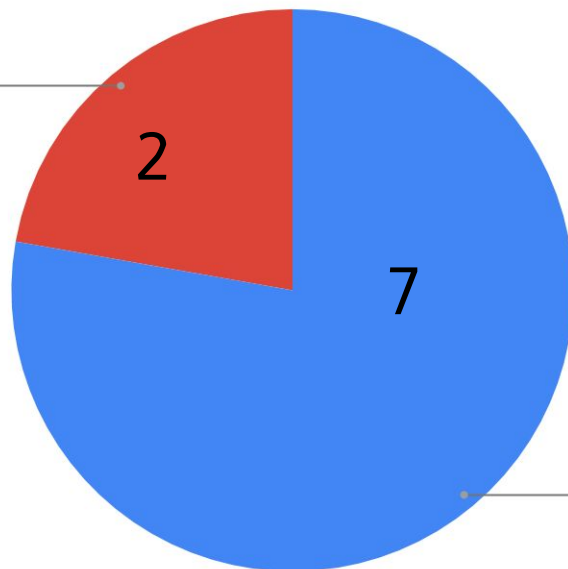
# Results

# Bugs by component

**Bugs by component**

**Optimizer Rule**

22.2%



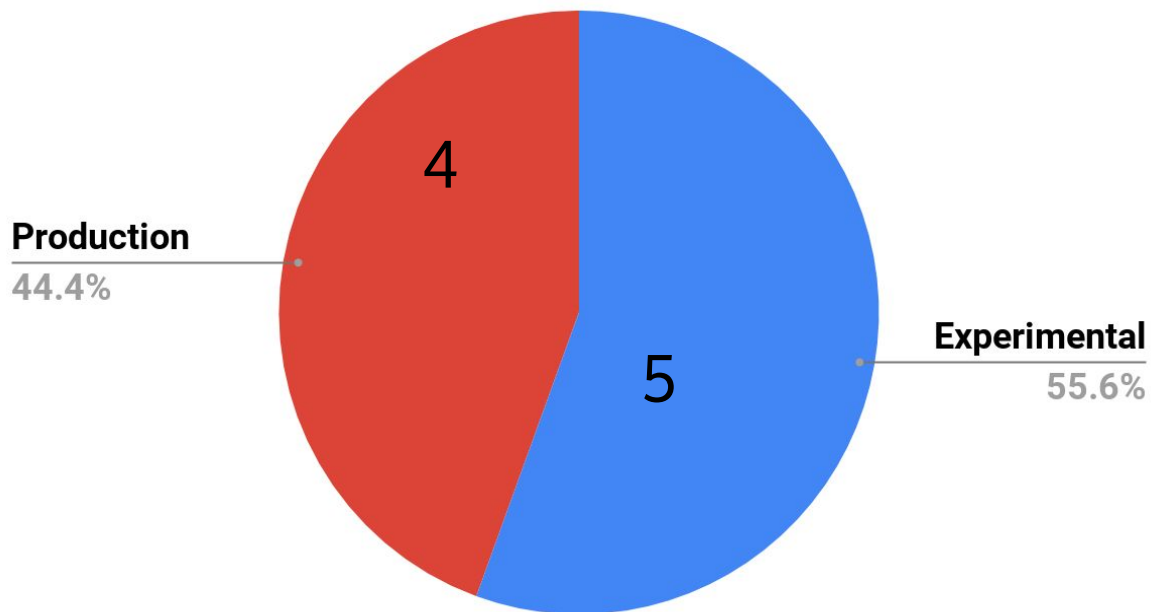
**Yul optimizer**

77.8%



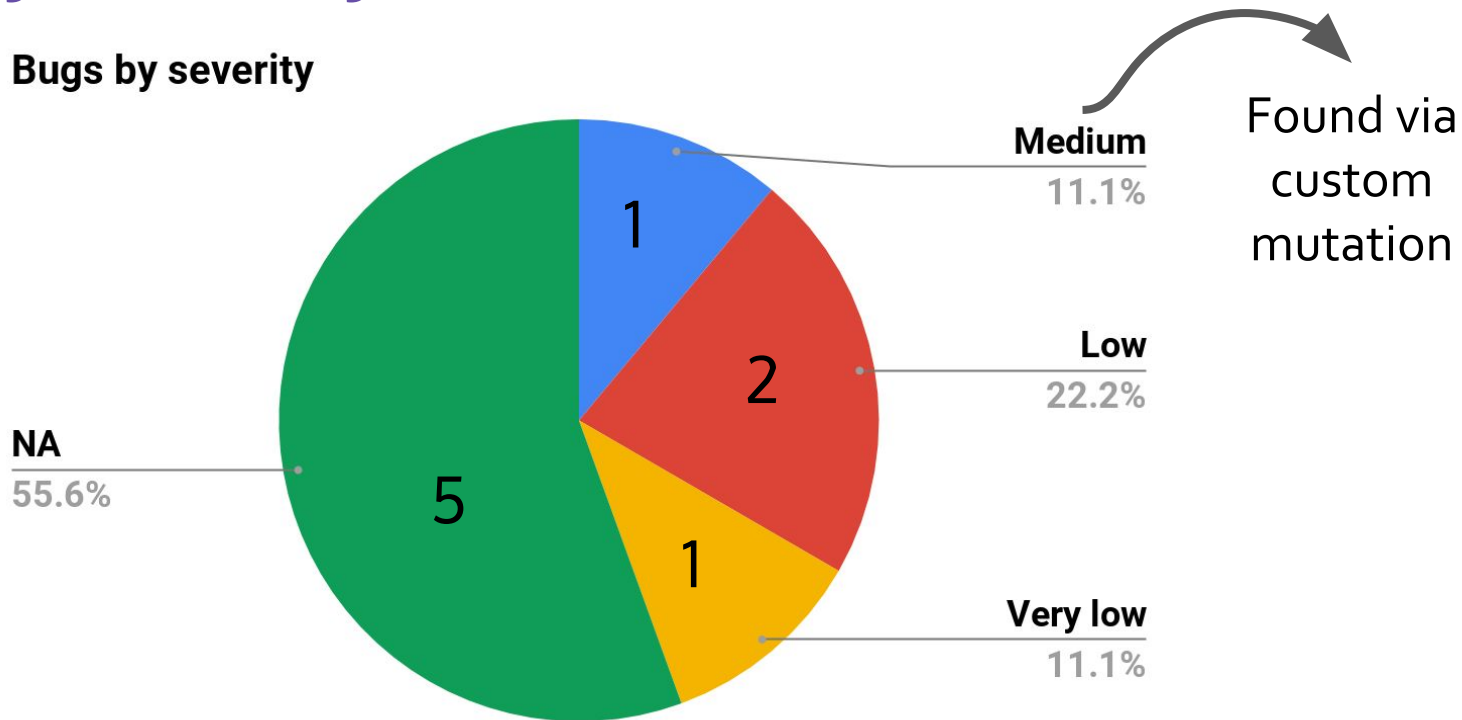
# Bugs by impact

**Bugs by impact**



# Bugs by severity

Bugs by severity





# Conclusion



# Conclusion

- Continuous structure-aware fuzzing for early bug discovery
- Useful for testing optimizer and data en/decoding
- Decent assurance
  - Evidence that it works
  - No formal guarantees though



Thank you!

Solidity talks @ EthCC3

Mathias and Eric:

What's New in Solidity,  
Day 1, Monge, 15:20

Chris: Metadata and  
Source Code Repository,  
Day 2, Monge, 15:55

