Fuzzing Smart Contract Systems Effectively

Valentin Wüstholz

CONSENSYS Diligence

Wish list for fuzzing contracts

1. Find bugs (that is, property violations)
2. Achieve high code coverage
3. Write as little test code as possible
4. Do so quickly :)

Good news: there are fuzzers to help you!

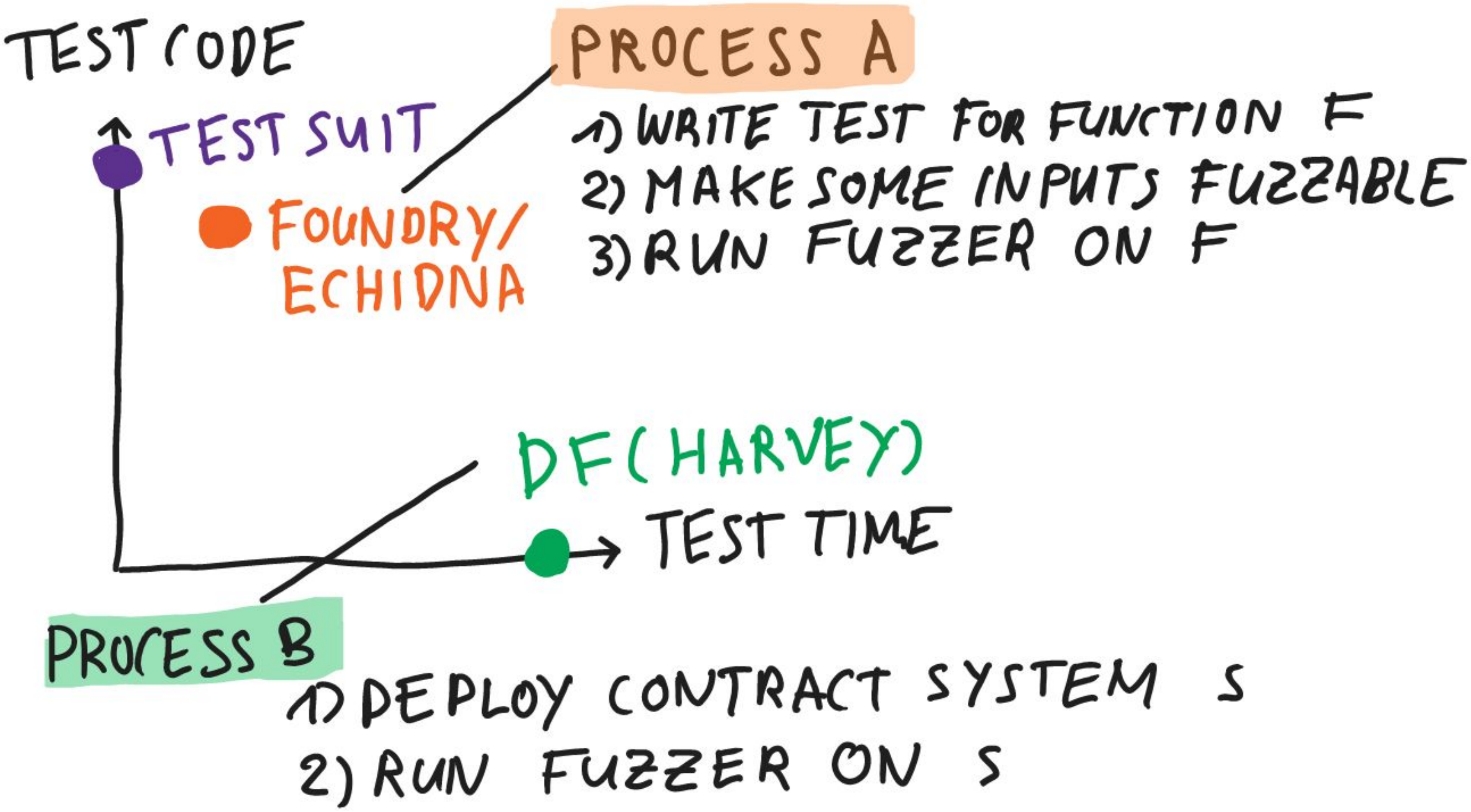
Bad news: mastering these can take a bit of time

Overview

1. The fuzzer spectrum
2. Fuzzing contract systems
3. How to increase coverage using fuzzing lessons
4. Testing your specifications and fuzzing setup
5. Fuzzing under the hood

The fuzzer spectrum

Fuzzing functions vs. fuzzing contract systems



PROCESS A

1) WRITE TEST FOR FUNCTION F
2) MAKE SOME INPUTS FUZZABLE
3) RUN FUZZER ON F

PROCESS B

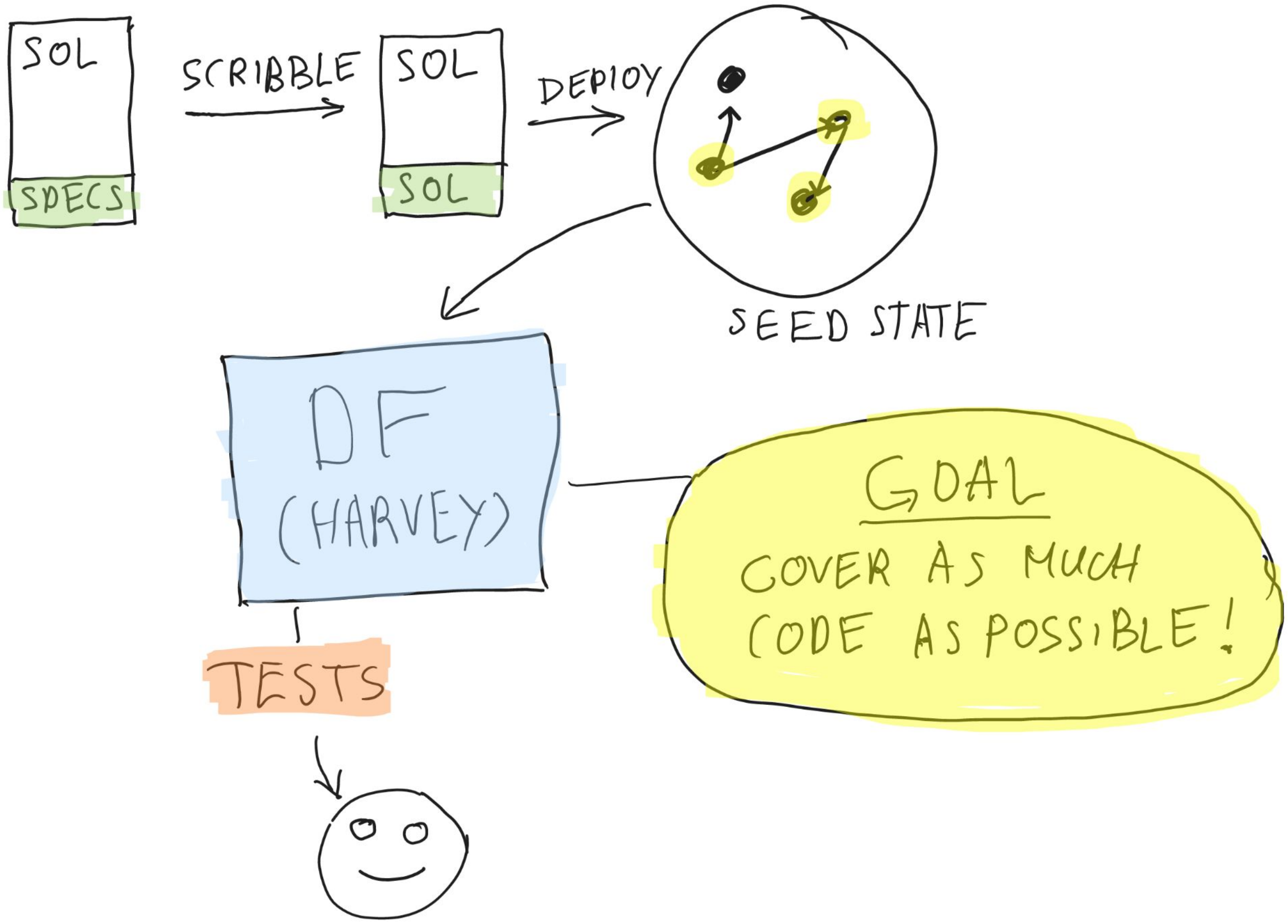
1) DEPLOY CONTRACT SYSTEM S
2) RUN FUZZER ON S

What makes fuzzing systems hard?

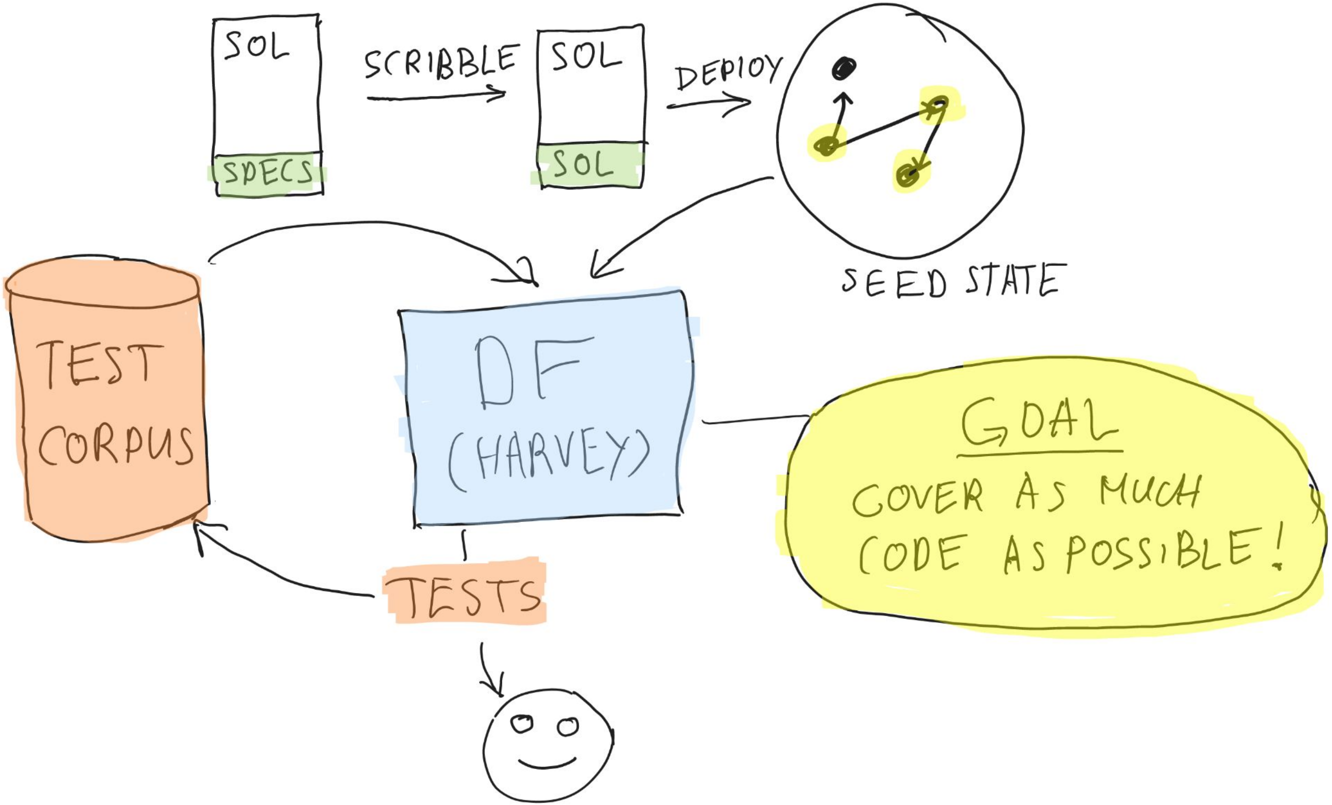
- Transactions can invoke **many contracts**
- Transactions can invoke **many functions**
- Functions can have **many inputs**
- **Grows exponentially** with number of transactions

Fuzzing contract systems

Typical workflows


SOL
SPECS
SCRIBBLE
SOL
SOL
DEPLOY
SEED STATE
DF
(HARVEY)
GOAL
COVER AS MUCH CODE AS POSSIBLE!
TESTS

Incremental fuzzing



Incremental fuzzing

- Enables **iterative changes** to code and specifications
- Provides **quick feedback**
- Reuses **existing corpus**

Typical workflow for new codebase

1) Set up fuzzer
2) Start fuzzing campaign
3) Write or refine properties (if necessary modify code)
4) Stop fuzzing campaign and review coverage, etc.
5) If necessary refine setup, goto (2)

How to increase coverage using fuzzing lessons

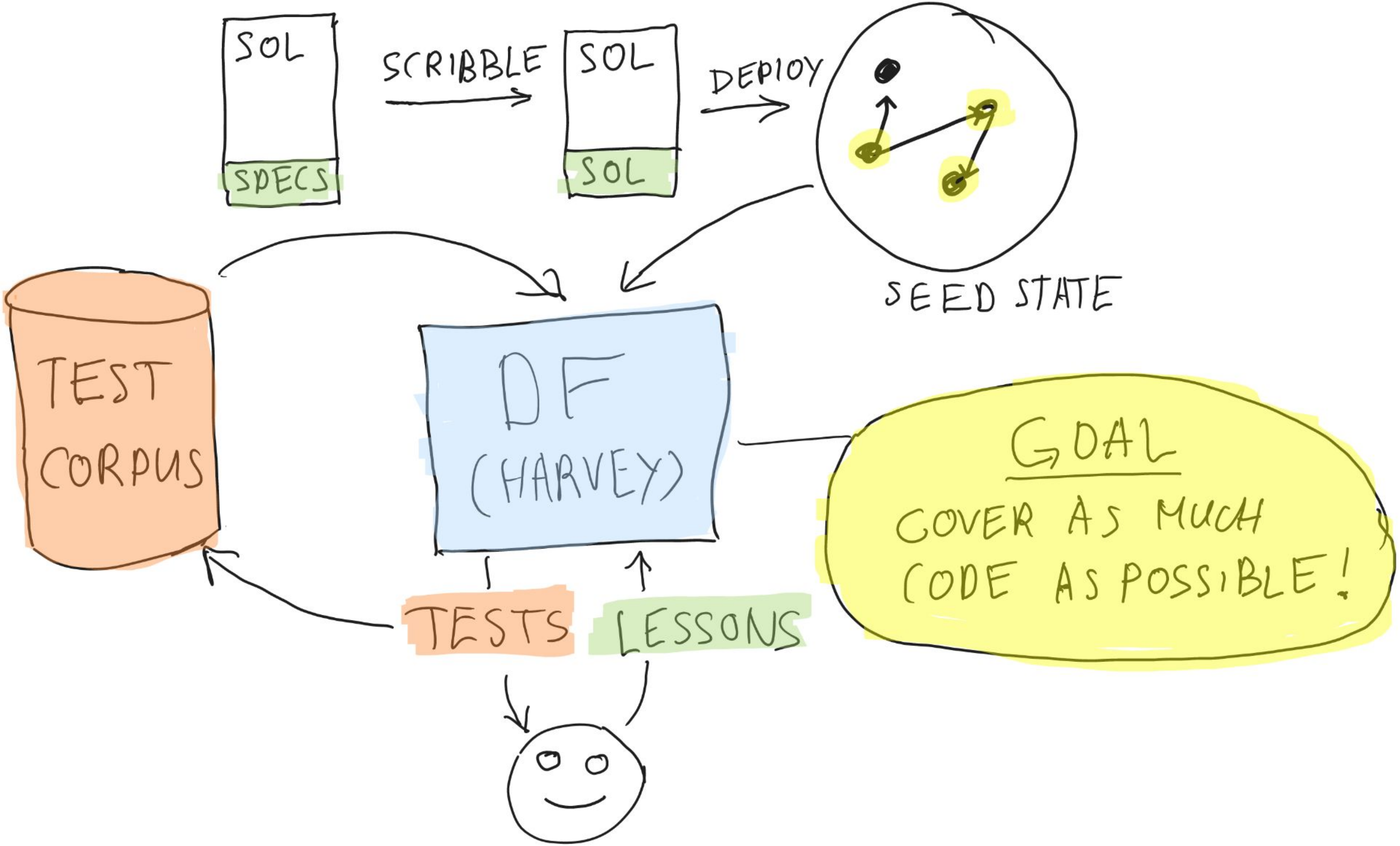
Helping the fuzzer one lesson at a time

Problem: some code isn’t covered

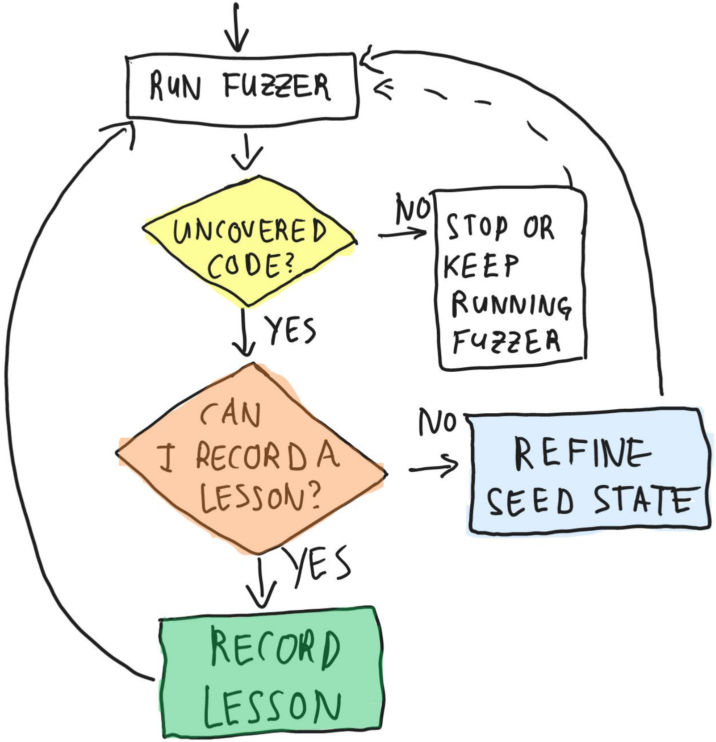
1. Make sure you have let the fuzzer run for some time
2. Review your fuzzer setup:
    1. What transactions are necessary to cover the code?
    2. Are all invoked contracts declared as “contracts under test”?
    3. Are preconditions for the transactions met? Missing token approvals?
    4. …

Fuzzing lessons: key idea

- Manually **suggest a sequence of transactions**
- **Concretely:** Write a script and let the fuzzer **observe the executed transactions**
- **Restart the fuzzer** with additional suggestions


SOL
SPECS
SCRIBBLE
SOL
SOL
DEPLOY
SEED STATE
TEST CORPUS
DF (HARVEY)
GOAL
COVER AS MUCH CODE AS POSSIBLE!
TESTS
LESSONS

Workflow for increasing coverage



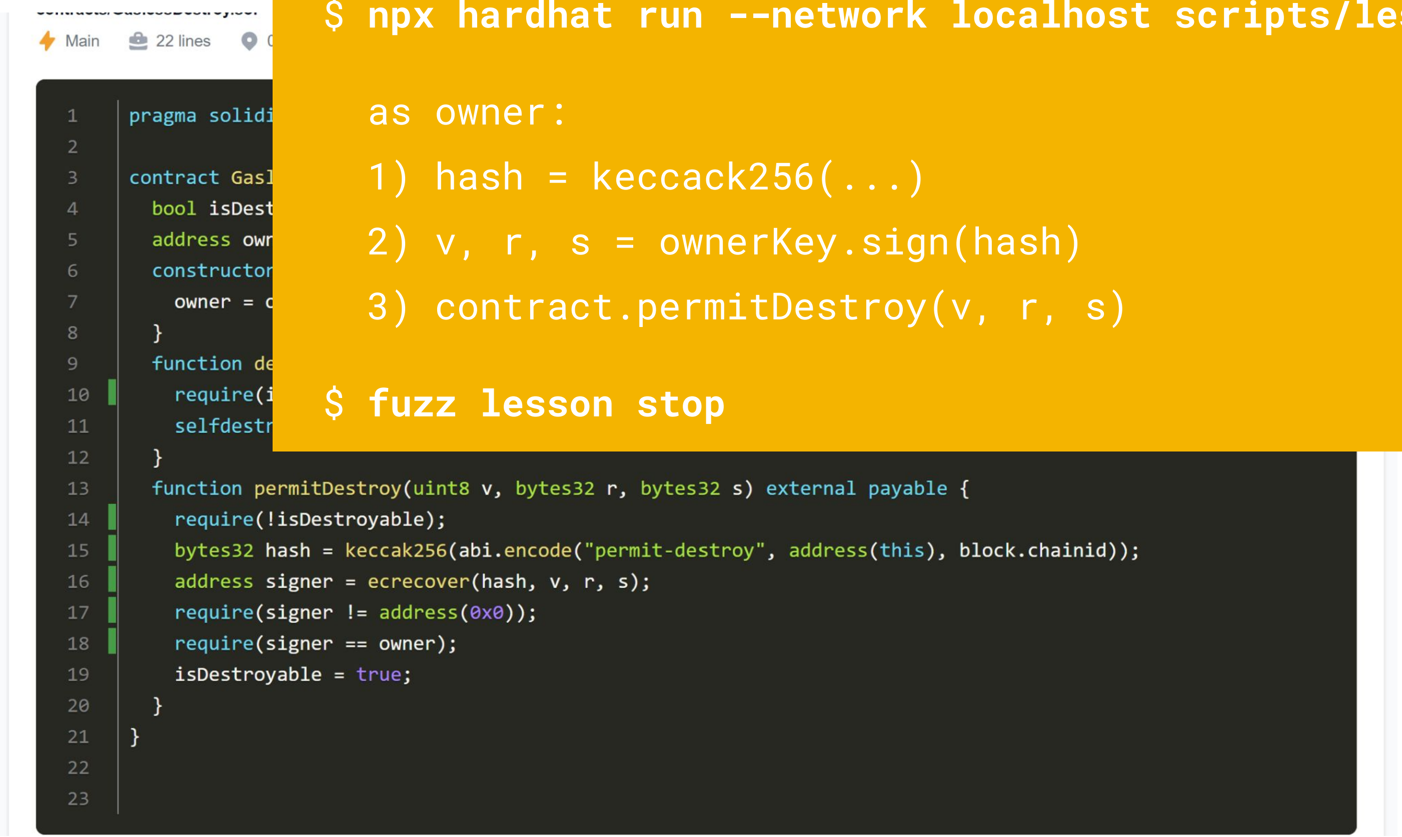
Example

```
$ fuzz lesson start --description "..."

$ npx hardhat run --network localhost scripts/lesson.js

  as owner:
  1) hash = keccack256(...)
  2) v, r, s = ownerKey.sign(hash)
  3) contract.permitDestroy(v, r, s)

$ fuzz lesson stop
```

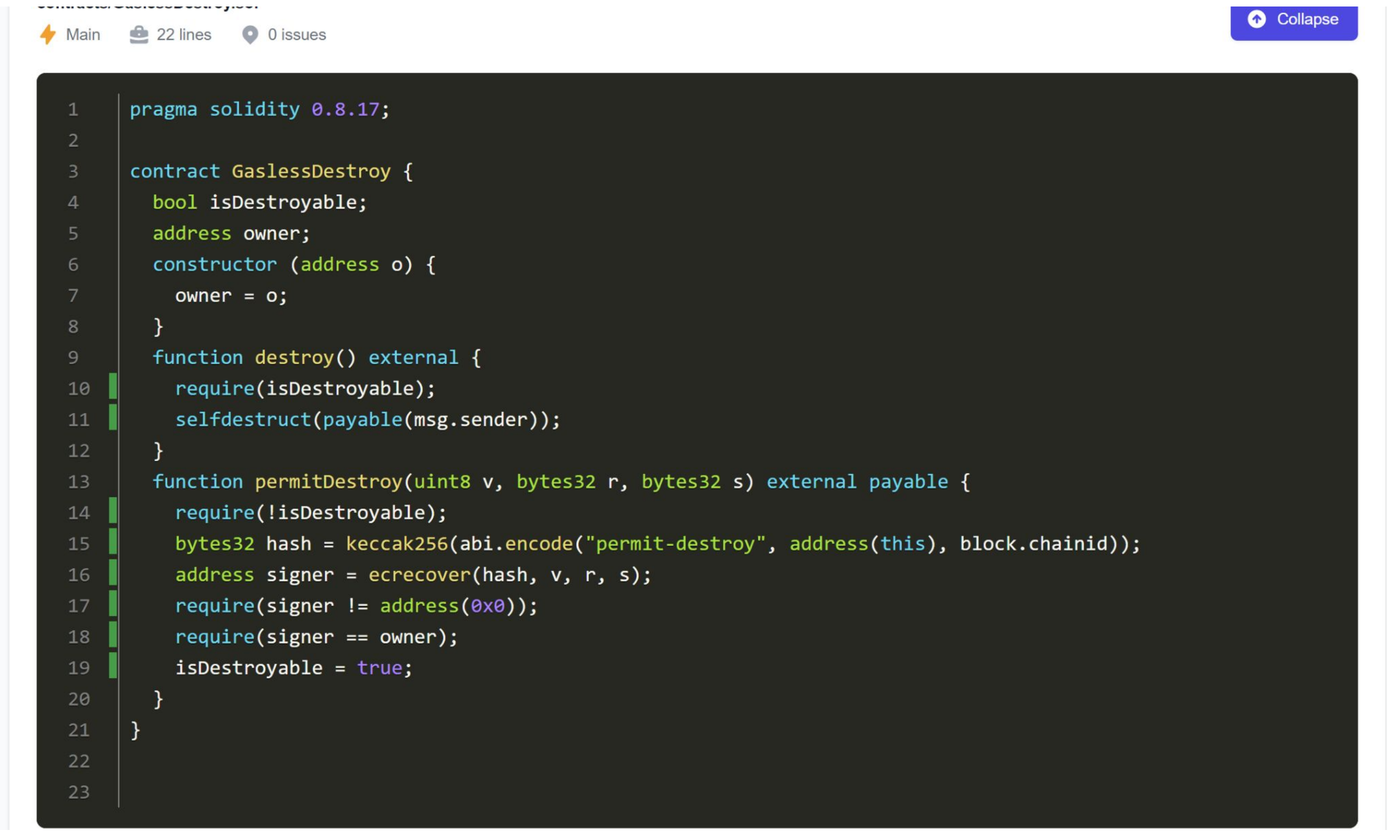


Main 22 lines 0

```
pragma solidi

contract Gasl
  bool isDest
  address own
  constructor
    owner = c
  }
  function de
    require(i
    selfdestr
  }
  function permitDestroy(uint8 v, bytes32 r, bytes32 s) external payable {
    require(!isDestroyable);
    bytes32 hash = keccak256(abi.encode("permit-destroy", address(this), block.chainid));
    address signer = ecrecover(hash, v, r, s);
    require(signer != address(0x0));
    require(signer == owner);
    isDestroyable = true;
  }
}
```

Example (after recording the lesson)



```solidity
pragma solidity 0.8.17;

contract GaslessDestroy {
  bool isDestroyable;
  address owner;
  constructor (address o) {
    owner = o;
  }
  function destroy() external {
    require(isDestroyable);
    selfdestruct(payable(msg.sender));
  }
  function permitDestroy(uint8 v, bytes32 r, bytes32 s) external payable {
    require(!isDestroyable);
    bytes32 hash = keccak256(abi.encode("permit-destroy", address(this), block.chainid));
    address signer = ecrecover(hash, v, r, s);
    require(signer != address(0x0));
    require(signer == owner);
    isDestroyable = true;
  }
}


```

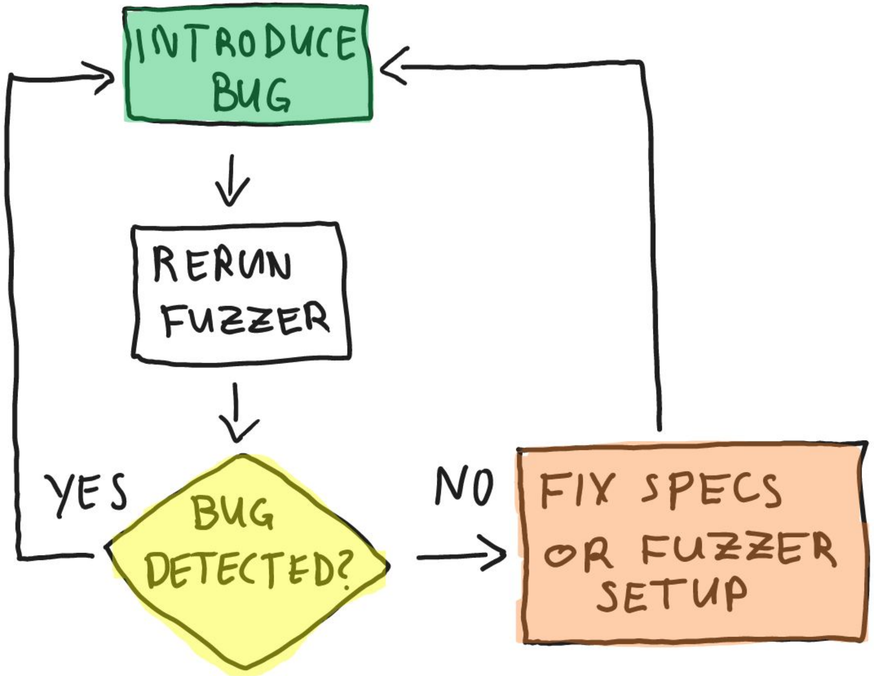
Testing specifications and fuzzing setup

A good specification catches bugs

Problem: how good are my specs?

- Related: **How good is my fuzzer setup?**
- Getting specifications “right” is challenging:
  - Code has corner cases, and so do specifications
  - **You can always write more specifications**
  - Tradeoff: **precision vs. effort**

Workflow for testing specifications



Example

```
/// #if_succeeds old(balanceOf(t)) <= balanceOf(t);
function transfer(address t, uint256 a) public {
    address owner = _msgSender();
    _transfer(owner, t, a);
    return true;
}
```

No bug!

Example: bug 1

```
/// #if_succeeds old(balanceOf(t)) <= balanceOf(t);
function transfer(address t, uint256 a) public {
    address owner = _msgSender();
    _transfer(owner, t, 1);
    return true;
}
```

No bug!

Example: refined specification 1

```
/// #if_succeeds old(balanceOf(t)) == balanceOf(t) + a;
function transfer(address t, uint256 a) public {
    address owner = _msgSender();
    _transfer(owner, t, a);
    return true;
}
```

Bug!

Example: refined specification 2

```
/// #if_succeeds t == msg.sender ||
///                old(balanceOf(t)) == balanceOf(t) + a;
function transfer(address t, uint256 a) public {
    address owner = _msgSender();
    _transfer(owner, t, a);
    return true;
}
```

No bug!

Fuzzing under the hood

Challenges when fuzzing smart contracts

Harvey

- Fuzzer for the **Ethereum Virtual Machine** (EVM) under development since **September 2017**
- Acquired by **ConsenSys** in 2018
- Integrated in **MythX** and **Diligence Fuzzing** products
- Regularly used in **audits** and **client engagements**
- Integrates **novel fuzzing techniques** developed in collaboration with Maria Christakis from MPI-SWS

What’s greybox fuzzing?

- Between **blackbox** (without feedback) and **whitebox** fuzzing (a.k.a. symbolic execution)
- Runs **concrete** inputs instead of **symbolic** inputs
- Uses **lightweight feedback** to guide fuzzer
- **No constraint solving**

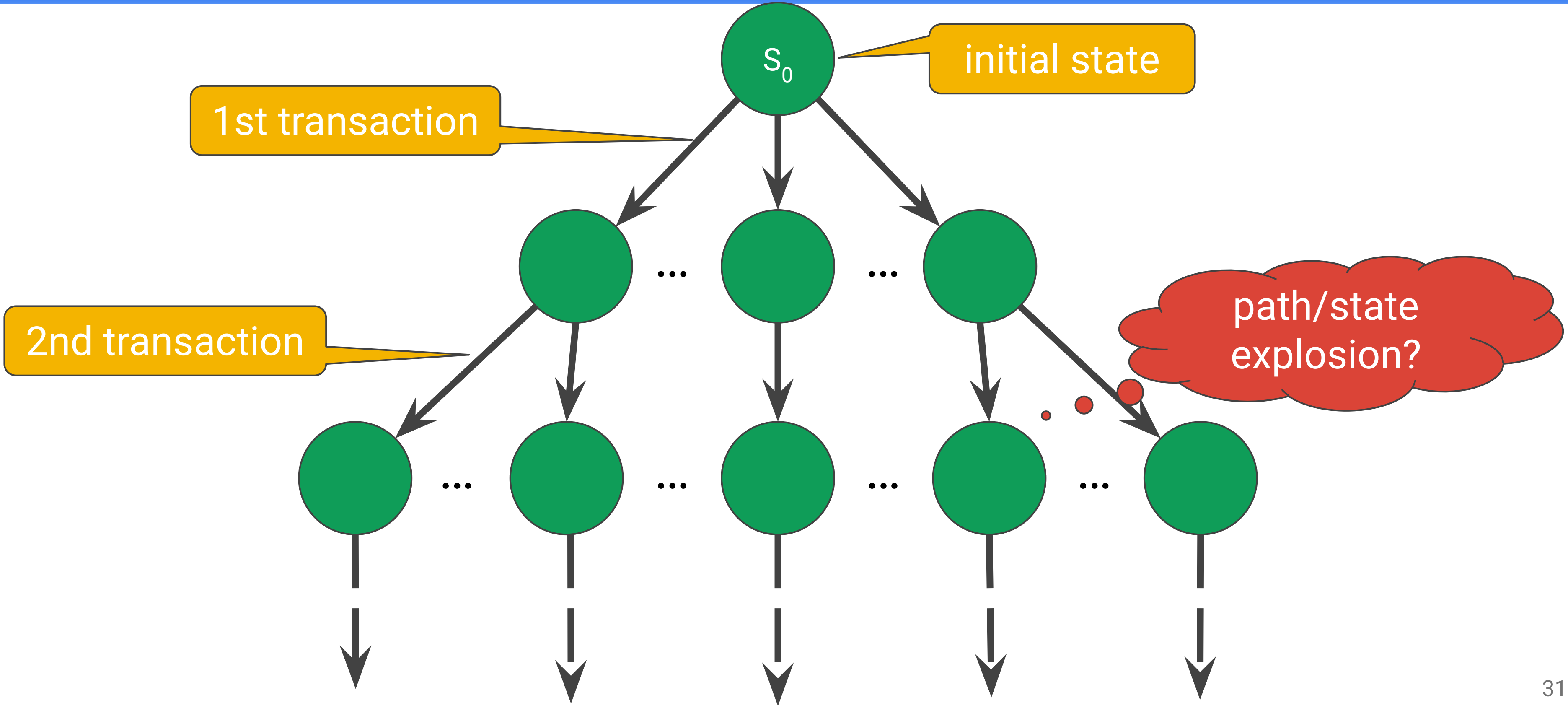
Greybox fuzzing algorithm

```
queue := run_seed_input(program, seed)
while (not interrupted) {
    i := select_input(queue); e := assign_energy(i)
    while (0 < e) {
        f := fuzz_input(i); path_id := run_input(program, f)
        if path_id not in queue { queue[path_id] := f }
        e := e - 1
    }
}
return queue
```

High-level idea for fuzzing contracts

- User provides **initial state** (incl. one or more contracts)
- User provides custom **correctness properties** (optional)
- Fuzzer **generates sequences of transactions** invoking contracts under test:
  - mutates transaction data (e.g., function inputs)
  - mutates sequences

Execution model



Fuzzing sounds simple, but ...

- **... efficient fuzzing (of contracts) is challenging!**
- Harvey incorporates several **novel fuzzing technique**
- Long-term research effort:

Valentin Wüstholz and Maria Christakis. **Harvey: A Greybox Fuzzer for Smart Contracts** (ESEC/FSE 2020)

Valentin Wüstholz and Maria Christakis. **Targeted Greybox Fuzzing with Static Lookahead Analysis** (ICSE 2020)

...

Challenge 1: narrow checks

```
function Bar(int256 a, int256 b, int256 c) returns (int256) {
    int256 d = b + c;
    if (d < 1) {
        if (b < 3) { return 1; }
        if (a == 42) { return 2; }
        return 3;
    } else {
        if (c < 42) { return 4; }
        return 5;
    }
}
```

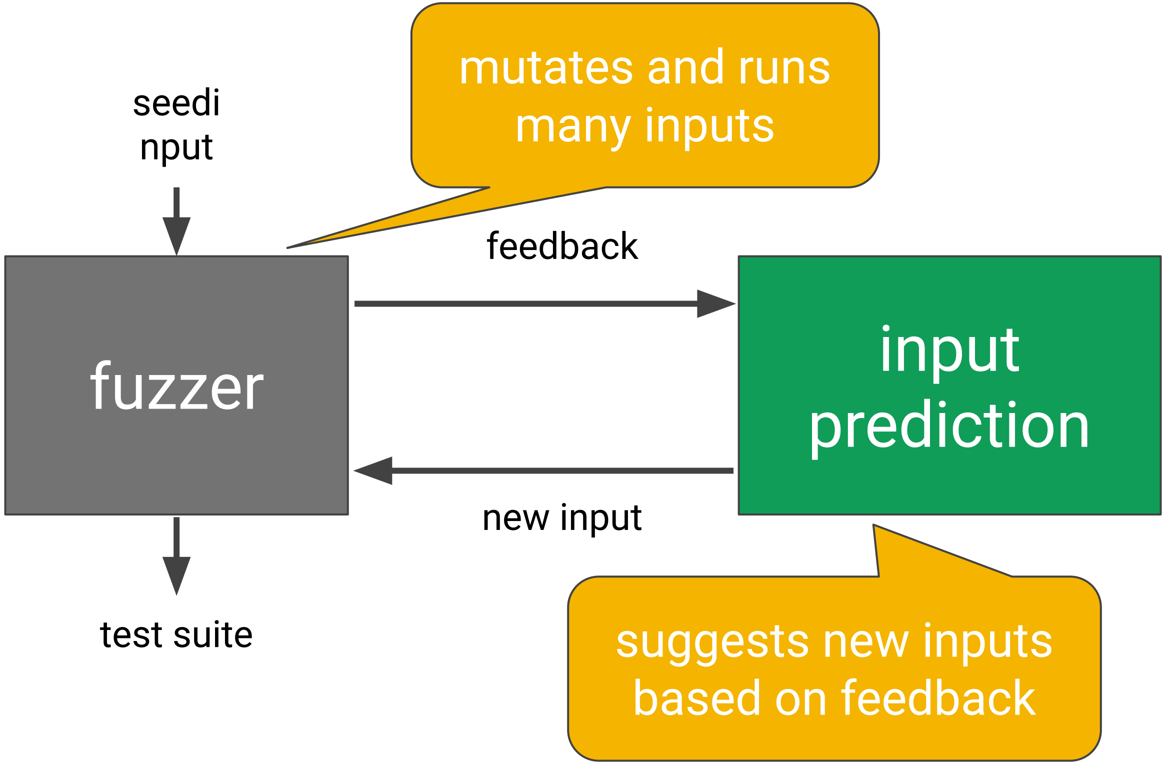
almost impossible to reach by random mutations

addressed by input prediction technique

Input prediction: key idea

- Make greybox fuzzing “more white” by **collecting additional feedback**
- **Approach:** use feedback to suggest new inputs
- **Feedback:** how far is the execution from “flipping” each branch (**branch distance**)

Fuzzing + Input Prediction = Coverage!


mutates and runs many inputs
seedi nput
feedback
fuzzer
input prediction
new input
test suite
suggests new inputs based on feedback

Branch distance: example

```
function Bar(int256 a, int256 b, int256 c) returns (int256) {
      int256 d = b + c;
      if (d < 1) {
            if (b < 3) { return 1; }
            if (a == 42) { return 2; }
            return 3;
      } else {
            if (c < 42) { return 4; }
            return 5;
      }
}
```

distance for **a==0**, b==3, c==-3?

42!

Branch distance: example **after fuzzing**

```
function Bar(int256 a, int256 b, int256 c) returns (int256) {
    int256 d = b + c;
    if (d < 1) {
        if (b < 3) { return 1; }
        if (a == 42) { return 2; }
        return 3;
    } else {
        if (c < 42) { return 4; }
        return 5;
    }
}
```

distance for **a==7**, b==3, c==-3?

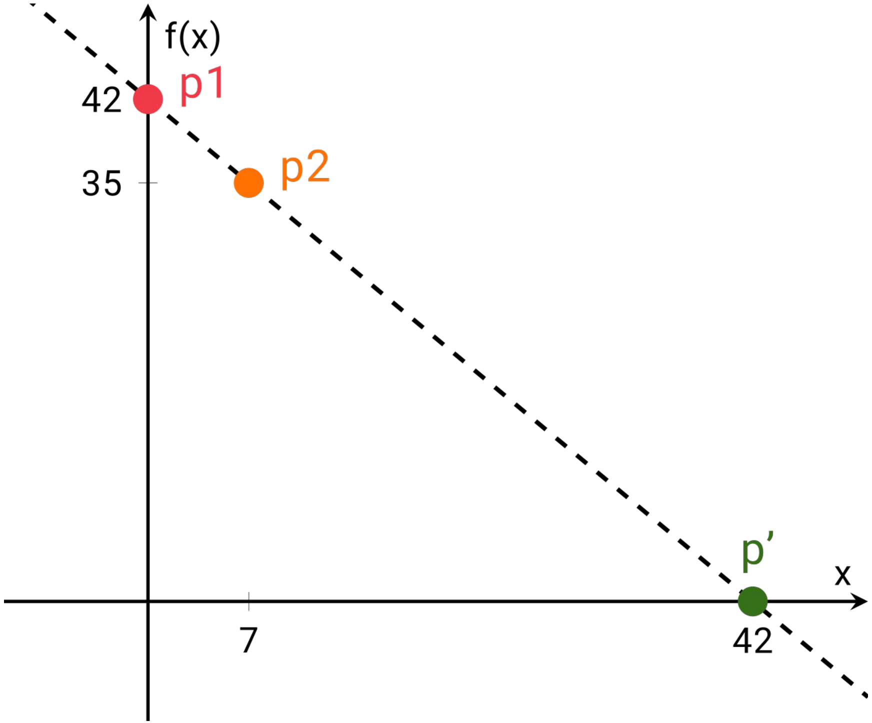
35!

What can we learn from the feedback?

- Input a **taints the branch**
- What input a’ should we try next?
  - What value a’ makes distance function **f(a’) == 0**?
  - Sounds familiar? Essentially **root-finding**!

Secant Method

- Iteratively finds roots
- **No derivatives needed!**



What if distance function isn’t linear?

- Apply **iteratively**!
- Works for non-linear conditions

  Example: **x**4 + x**2 == …**
- **Fun fact: one step is often enough**

  (average success rate of **99%** in our experiments)

Effectiveness of input prediction

- Finds **bugs** orders of magnitude faster (**~5x**)
- Increases **coverage** significantly (up to **~3x**)

Challenge 2: deep vulnerabilities

```
contract Foo {
  int256 private x;
  int256 private y;
  function Bar() public { assert(x != 42); }
  function SetY(int256 ny) public { y = ny; }
  function IncX() public { x++; }
  function CopyY() public { x = y; }
}
```

persistent state variables

requires multiple calls to trigger

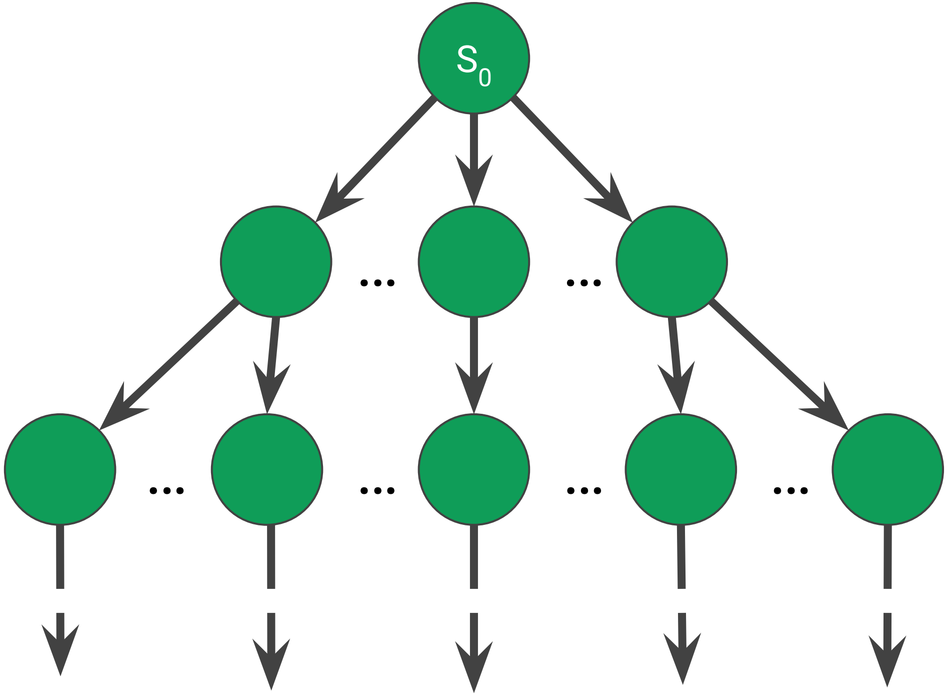
addressed by demand-driven fuzzing

Demand-driven fuzzing: key idea

- Fuzzing long transaction sequences is expensive:
  - More inputs to fuzz
  - Longer execution
  - Number of paths grows exponentially
- **Idea**: avoid it unless it may improve coverage

Coverage of what?

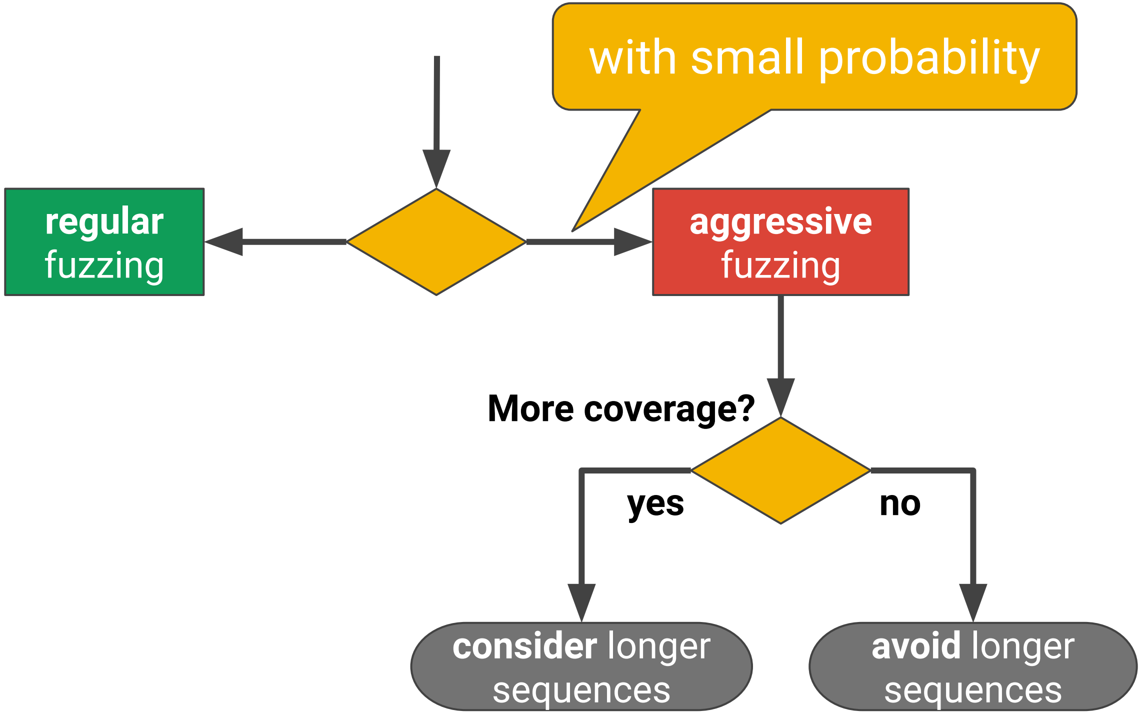
- **Issue**: increasing coverage is trivial if paths span multiple transactions
- Harvey only considers **path of last transaction**



Aggressive fuzzing

- Use aggressive fuzzing to determine if coverage may increase and longer sequences should be considered
- Main difference: **allow fuzzing the persistent state**

Regular vs. aggressive fuzzing



with small probability

regular fuzzing

aggressive fuzzing

More coverage?

yes

no

consider longer sequences

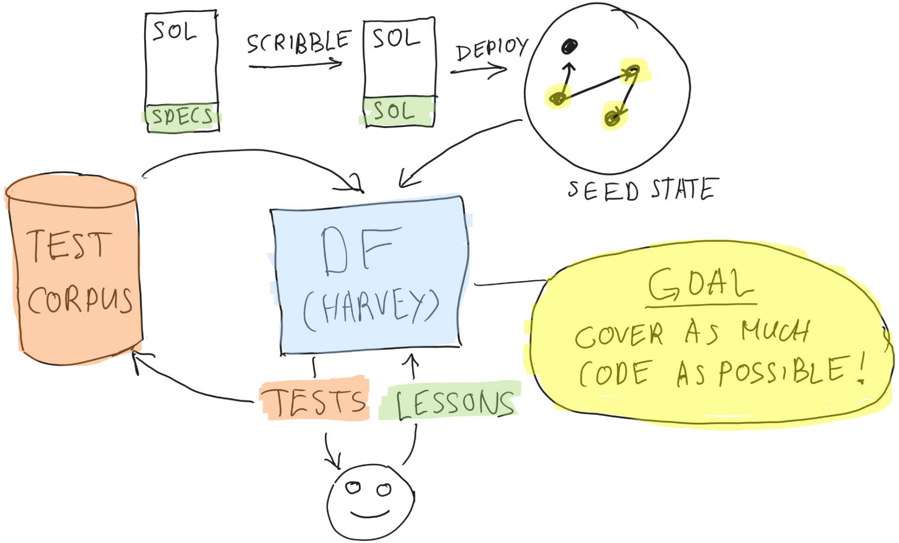
avoid longer sequences

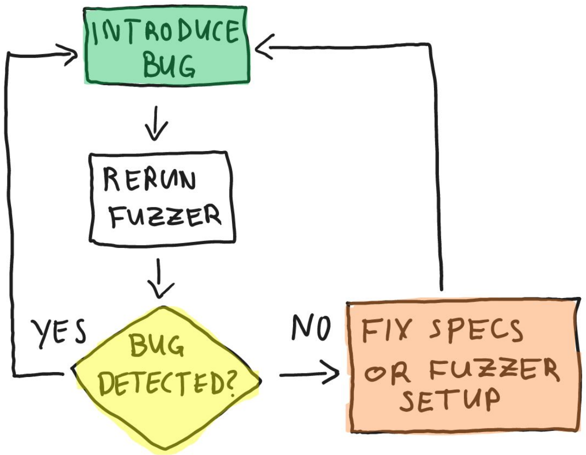
Effectiveness of demand-driven fuzzing

- **~60%** of bugs require multiple transactions
- Finds **bugs** orders of magnitude faster (**~3x**)
- Increases **coverage** significantly

Summary

Shared some tips and tricks for fuzzing smart contract systems effectively


SOL
SPECS
SCRIBBLE
SOL
SOL
DEPLOY
SEED STATE
TEST CORPUS
DF (HARVEY)
GOAL
COVER AS MUCH CODE AS POSSIBLE!
TESTS
LESSONS


INTRODUCE BUG
RERUN FUZZER
BUG DETECTED?
YES
NO
FIX SPECS OR FUZZER SETUP