Automated Program Analysis for Smart Contracts

Valentin Wüstholz (ConsenSys Diligence/MythX)

What are smart contracts?

- Programs managing accounts on a blockchain
- Main purpose: can replace trusted third party
- Examples: games, auctions, asset management

Ethereum

- Blockchain launched in 2015
- Two types of accounts with unique address:
    - Users
    - Contracts (incl. code and persistent storage)
- Introduces support for expressive smart contracts
- Uses VM to execute bytecode of contracts
- High-level languages that compile to EVM bytecode: Solidity, Vyper, ...

Execution Model

```
sender, args, block := *, *, *
c := create_contract(sender, args, block)  // creation transaction
while * {
    sender, args, block := *, *, *        // subsequent transaction
    assume valid_block(block)             // e.g., block number increases
    c.invoke(sender, args, block)
}
```

Some Challenges

- Unbounded number of paths
- Unbounded number of users
- Cryptographic operations (e.g., hashing)
- Modeling of persistent storage (unbounded map)
- Limited control over execution order
- ...

Realization: difficult to support in single analyzer

Our solution: MythX analysis service

- Integrates multiple analysis components:
  - **Mythril**: symbolic execution
  - **Harvey**: greybox fuzzing
  - **Maru**: light-weight static analysis and linting
- More than 3 Million analyzed contracts in 2019
- More than 100’000 issues
- Easy to integrate in client tools

MythX Demo

Overview of remaining talk

1. Harvey greybox fuzzer

2. Targeted fuzzing using static lookahead analysis

Harvey Greybox Fuzzer

Joint work with Maria Christakis (MPI-SWS)

Fuzzing

- Used for automated test generation
- Wide range of techniques:
  - Blackbox: QuickCheck, ...
  - Greybox: AFL, libfuzzer, ...
  - Whitebox (AKA dynamic symbolic execution): SAGE, Cute, KLEE, …
- Come with different effectiveness-efficiency tradeoffs

Blackbox fuzzing

1. Generate random input:
   - based on input type
   - based on input specification (e.g., grammar)
   - based on mutations of seed file(s)
2. Run input and check for crash or property violation

Typically high efficiency, but low effectiveness

Whitebox fuzzing

1. Run input and check for crash or property violation
2. Collect path constraint
3. Flip a branch to explore a new path
4. Solve constraints to generate corresponding input

Typically low efficiency, but high effectiveness

Greybox fuzzing

1. Generate random input (typically by mutating existing one)
2. Run input and check for crash or property violation
3. Add to queue if new path is explored (light-weight instrumentation)

Typically medium efficiency and medium effectiveness

Greybox fuzzing algorithm

```
TS := run_seed_inputs(P, S)
while (not interrupted) {
    i := select_input(TS); e := assign_energy(i)
    while (0 < e) {
        f := fuzz_input(i); pid := run_input(P, f)
        if pid not in TS { TS[pid] := f } e := e - 1
    }
}
```

Path identifiers

- Naive solution: hash program locations along execution
- Issue: not efficient and too many paths
- Harvey takes inspiration from AFL:
  - Count basic block transitions using large array
  - Hash the ($\log_2$) counts in the array

Harvey

- Greybox fuzzer for the Ethereum Virtual Machine (EVM)
- Under development since September 2017
- Acquired by ConsenSys in 2018 and now part of MythX
- Initially: apply greybox fuzzing to transaction sequences
- Several novel extensions:
  - Input prediction technique
  - Demand-driven sequence fuzzing

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

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; }

}
```

Requires multiple transactions/calls to trigger

Addressed by demand-driven fuzzing

Input prediction: key idea

- Make greybox fuzzing slightly “whiter”
- Approach: instrument program to record branch distance trace
- Branch distance: how far is the execution from “flipping” the branch

Branch distance: example

```
function Bar(int256 a, int256 b, int256 c) public returns (int256) {
    int256 d = b + c;
    if (d < 1) {
        f (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) public returns (int256) {
    int256 d = b + c;
    if (d < 1) {
        f (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?

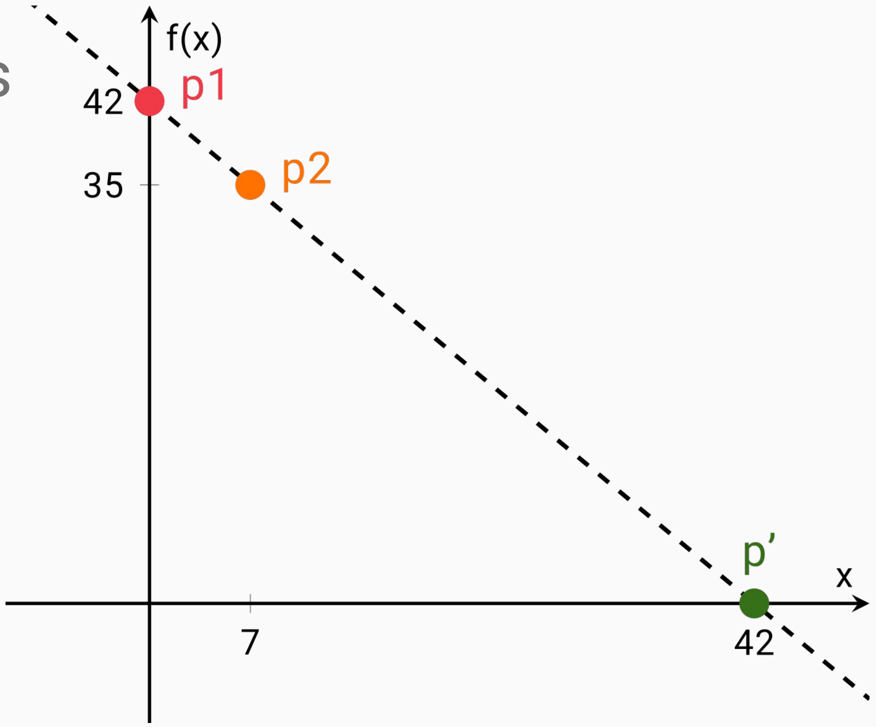
35!

Can we learn something interesting?

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

Secant Method

- For iteratively finding roots
- No derivatives needed



What if distance function isn’t linear?

- Apply iteratively!
- Works for non-linear conditions, e.g., x**4 + x** 2 == …
- Fun fact: one step is often enough

Effectiveness of input prediction

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

Demand-driven fuzzing: key idea

- Fuzzing long transaction sequences is expensive:
  - More inputs to fuzz
  - Longer execution
  - Number of paths grows exponentially
- **Idea**: don’t do it unless it may improve coverage

Coverage of what?

- Issue with considering a path to span multiple transactions: increasing coverage is trivial
- Harvey only considers the path of last transaction

Aggressive fuzzing

- Use more aggressive fuzzing to determine if longer sequences should be considered
- Aggressive fuzzing: allow fuzzing the persistent state
- Enable aggressive fuzzing occasionally:
  - Increases coverage?
  - If not: no need for longer sequences

Custom mutation operators

1. Fuzz inputs of transaction t
2. Insert transaction t’ before transaction t
3. Replace transactions before t with another sequence

To avoid regenerating “interesting” sequences and individual transactions: populate and use pools

Effectiveness of demand-driven fuzzing

- ~80% of bugs require multiple transactions
- Finds bugs orders of magnitude faster (~3x)
- Increases coverage significantly

Targeted Fuzzing using Static Lookahead Analysis

Joint work with Maria Christakis (MPI-SWS)

To appear at ICSE ‘20

Targeted fuzzing: key idea

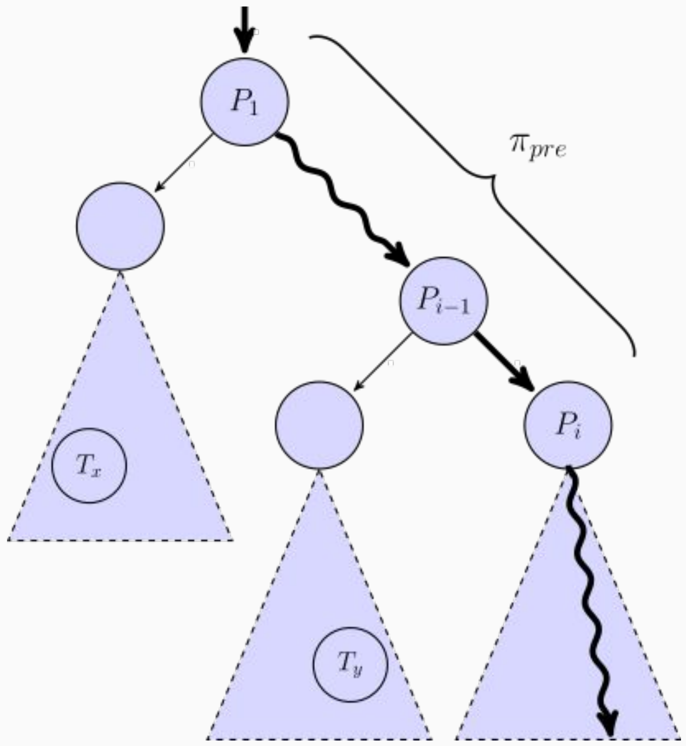
- Identify target program locations:
  - Examples: newly added code, static analysis warnings, ...
- Focus fuzzing on executions that reach target locations

Lookahead analysis

- Problem for fuzzers: can’t predict which inputs are “close” to targets
- Solution: **static** “lookahead analysis”
- Run online for each input that is added to the queue

Lookahead analysis for path π

- Computes set of split points $\{P_1, \dots, P_i\}$
- Computes no-target-ahead prefix π_{pre}



Trade-offs

- Partially path-sensitive: increases precision
- Fuzzer guarantees feasibility of prefix
- Static analysis needs to be efficient: limits precision

Bran: abstract interpreter for EVM

- Performs abstract interpretation on the bytecode
- Very light-weight by using constant propagation
- Used to perform lookahead analysis:
    - Compute postcondition ϕ of prefix π_{pre} ending at "split point" P_i
    - Check for reachability of target locations starting from P_i assuming ϕ

Greybox fuzzing algorithm

```
TS := run_seed_inputs(P, S)

while (not interrupted) {

     i := select_input(TS); e := assign_energy(i)

     while (0 < e) {

          f := fuzz_input(i); pid := run_input(P, f)

          if pid not in TS { TS[pid] := f } e := e - 1

     }

}
```

Targeted greybox fuzzing algorithm

```
TS := run_seed_inputs(P, S)
while (not interrupted) {
     i := select_input(TS); e := assign_energy(i)
     while (0 < e) {
          f := fuzz_input(i); pid := run_input(P, f)
          if pid not in TS { TS[pid] := f } e := e - 1
     }
}
```

Assign based on lookahead analysis

Run lookahead analysis for input

Assigning energy

- Inputs in queue may share same no-target-ahead prefix
- Essentially partitions the queue
- Assign more energy if:
  - input’s no-target-ahead prefix was rarely executed
  - or any of its split points was rarely executed

Rare no-target-ahead prefix

- Inspired by FairFuzz
- Find no-target-ahead prefix with fewest executions (e)
- Rare if executed < $2^{\mathrm{ceil}(\log2(e))}$
- Example: e == 23 ⇒ rare if executed fewer than 32 times

Evaluation: effectiveness

- Reaches 83% of challenging target locations significantly faster
- 14x median speedup
- Lookahead analysis is very light-weight (~3s per fuzzing campaign)

Other ongoing work

- Verification of smart contracts
- Specification language
- Mutation testing framework: https://github.com/JoranHonig/vertigo

Summary

- Gave overview of our work on smart contract analysis
- Two concrete research topics:
  - Effective greybox fuzzing of smart contracts
  - Targeted fuzzing using static lookahead analysis

MythX

Give MythX a try at mythx.io!