TRAIL OF BITS

Before starting

- git clone https://github.com/crytic/building-secure-contracts
- git checkout dappcon-2022

Building secure contracts: How to fuzz like a pro

Who are we?

- **Troy Sargent (@0xalpharush)**
- **Josselin Feist (@montyly)**

- **Trail of Bits: trailofbits.com**
  - We help developers to build safer software
  - R&D focused: we use the latest program analysis techniques
  - Slither, Echidna, Tealer, Amarna, solc-select, ..

Agenda

- How to find bugs?
- What is property based testing?
- Exercises
- How to define good invariants?
- Comparison with similar tools

How to Find Bugs?

```
/// @notice Allow users to buy token. 1 ether = 10 tokens
/// @param tokens The numbers of token to buy
/// @dev Users can send more ether than token to be bought, to give gifts to the
team.
function buy(uint tokens) public payable{
    _valid_buy(tokens, msg.value);
    _mint(msg.sender, tokens);
}


/// @notice Compute the amount of token to be minted. 1 ether = 10 tokens
/// @param desired_tokens The number of tokens to buy
/// @param wei_sent The ether value to be converted into token
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```

How to Find Bugs?

- **4 main techniques**
  - Unit tests
  - Manual analysis
  - Fully automated analysis
  - Semi automated analysis

How to Find Bugs?

- **Unit tests**
    - Benefits
        - Well understood by developers
    - Limitations
        - Mostly cover “happy paths”
        - Might miss edge cases

How to find bugs?

```
function test_buy(uint256 tokens_to_receive, uint256 ether_to_send) public {

    uint256 pre_buy_balance = token.balanceOf(address(this));

    mock.buy.call{value: ether_to_send}(tokens_to_receive);

    assert(token.balanceOf(address(this)) == pre_buy_balance + tokens_to_receive)

}
```

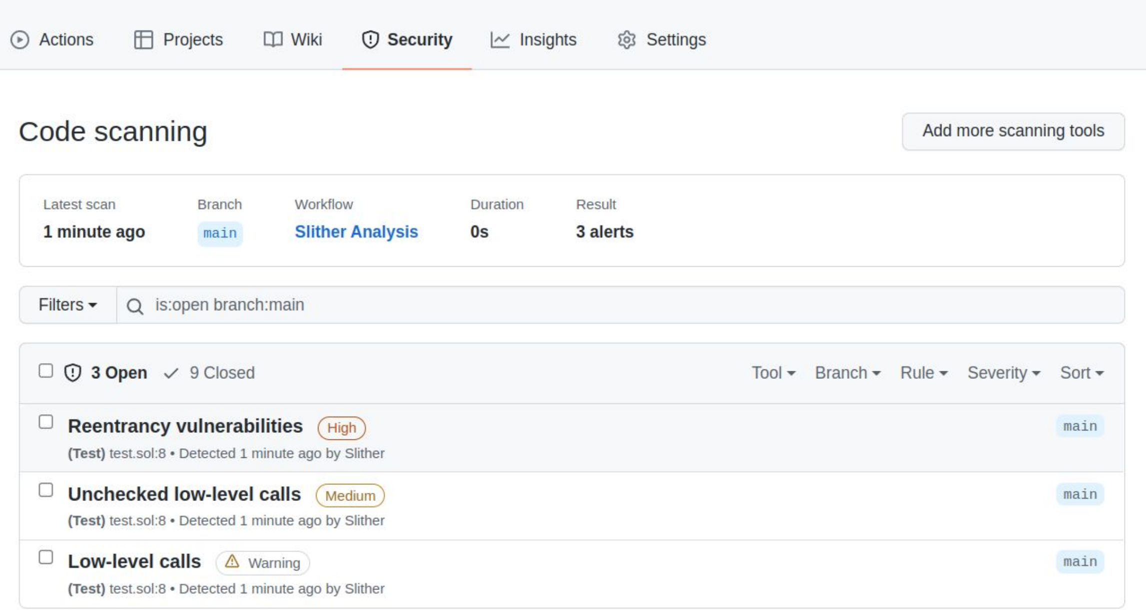
How to Find Bugs?

- **Manual review**
    - Benefits
        - Can detect any bug
    - Limitations
        - Time consuming
        - Require specific skills
        - Does not track code changes
    - Ex: Security audit

How to Find Bugs?

- **Fully automated analysis**
  - Benefits
    - Quick & easy to use
  - Limitations
    - Cover only some class of bugs
  - Ex: Slither

Slither Action



Code scanning

Latest scan	Branch	Workflow	Duration	Result
1 minute ago	main	Slither Analysis	0s	3 alerts

is:open branch:main

□ 3 Open ✓ 9 Closed

□ **Reentrancy vulnerabilities** High — main
(Test) test.sol:8 • Detected 1 minute ago by Slither

□ **Unchecked low-level calls** Medium — main
(Test) test.sol:8 • Detected 1 minute ago by Slither

□ **Low-level calls** ⚠ Warning — main
(Test) test.sol:8 • Detected 1 minute ago by Slither

How to Find Bugs?

- **Semi automated analysis**
  - Benefits
    - Great for logic-related bugs
  - Limitations
    - Require human in the loop
  - Ex: Property based testing with Echidna (today's topic)

What is property based testing?

Fuzzing

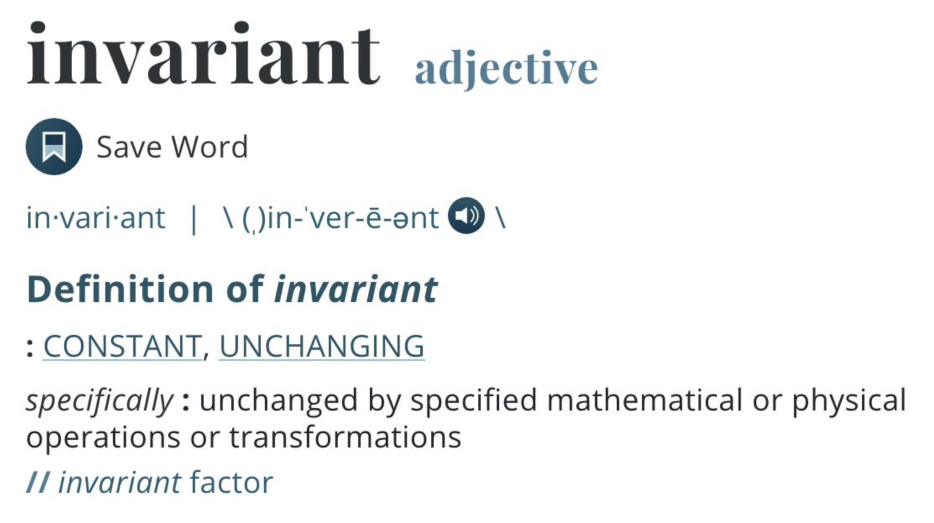
- **Stress the program with random inputs**
  - Most basic fuzzer: randomly type on your keyboard
- **Fuzzing is well established in traditional software security**
  - AFL, Libfuzzer, go-fuzz, ..

Property based testing

- **Traditional fuzzers generally detect crashes**
  - Smart contracts don’t (really) have crashes
- **Property based testing**
  - User defines invariants
  - Fuzzer generates random inputs
  - Check whether specified “incorrect” state can be reached
  - “Unit tests on steroids”

Invariant

- **Something that must always be true**



invariant adjective

Save Word

in·vari·ant | \ (ˌ)in-ˈver-ē-ənt \

Definition of *invariant*

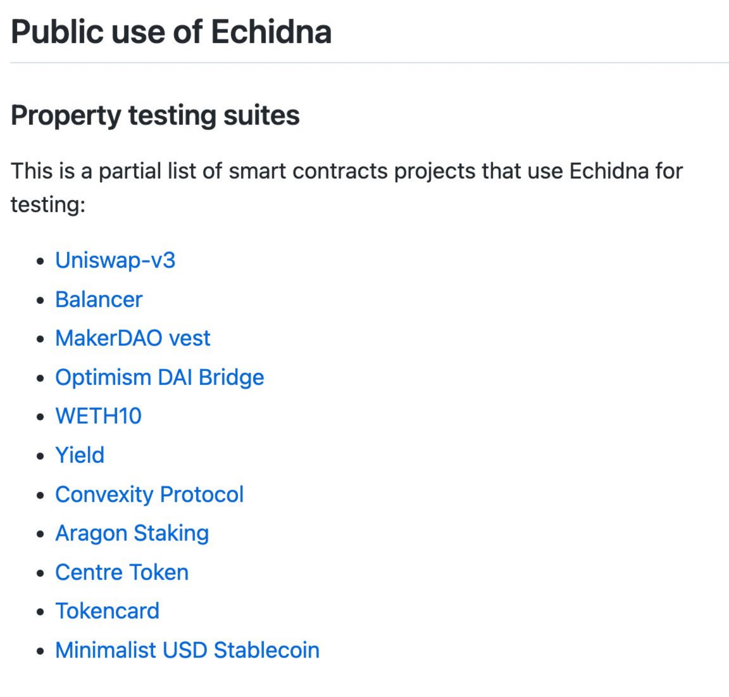
: CONSTANT, UNCHANGING

specifically : unchanged by specified mathematical or physical operations or transformations

// *invariant* factor

Echidna

- **Smart contract fuzzer**
- **Open source: github.com/crytic/echidna**
- **Heavily used in audits & mature codebases**



Public use of Echidna

Property testing suites

This is a partial list of smart contracts projects that use Echidna for testing:

- Uniswap-v3
- Balancer
- MakerDAO vest
- Optimism DAI Bridge
- WETH10
- Yield
- Convexity Protocol
- Aragon Staking
- Centre Token
- Tokencard
- Minimalist USD Stablecoin

Invariant - Token’s total supply

```
pragma solidity 0.7.0;

contract Token{

   mapping(address => uint) public balances;

   function transfer(address to, uint value) public{

        balances[msg.sender] -= value;

        balances[to] += value;

    }

 }
```

Invariant - Token’s total supply

User balance never exceeds total supply

Exercises

Exercise 1

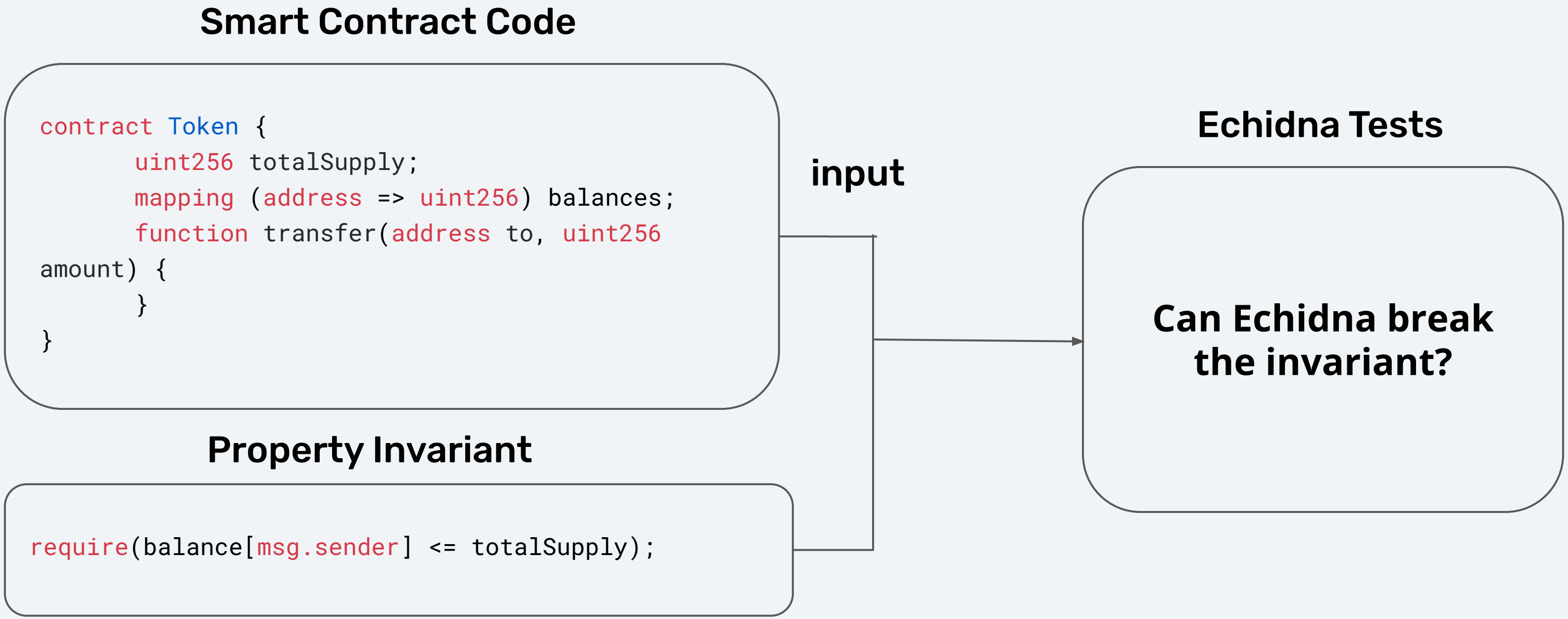
- program-analysis/echidna/Exercise-1.md
- Exercise-1.md
- Goal: check for correct arithmetic
- Note: use Solidity 0.7 (see solc-select if needed)

First: try without the template!

Exercise 1 - Template

```
contract TestToken is Token {

   address echidna_caller = msg.sender;



   constructor() public {
       balances[echidna_caller] = 10000;
   }


   // add the property

}
```

Exercise 1 - Solution



Smart Contract Code

```
contract Token {
        uint256 totalSupply;
        mapping (address => uint256) balances;
        function transfer(address to, uint256
amount) {
        }
}
```

Property Invariant

```
require(balance[msg.sender] <= totalSupply);
```

Echidna Tests

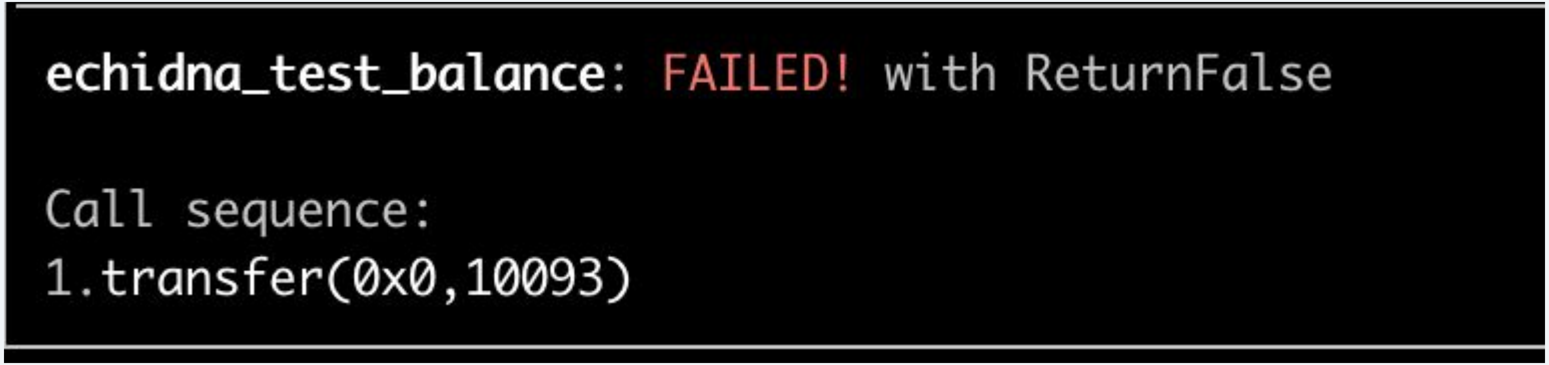
Can Echidna break the invariant?

Exercise 1 - Solution

```
contract TestToken is Token {

    address echidna_caller = msg.sender;



    constructor() public {
        balances[echidna_caller] = 10000;
    }


    function echidna_test_balance() view public returns(bool) {
        return balances[echidna_caller] <= 10000;
    }
}
```

Exercise 1 - Solution

```
$ echidna-test solution.sol
```



```
echidna_test_balance: FAILED! with ReturnFalse

Call sequence:
1.transfer(0x0,10093)
```

Exercise 2

- program-analysis/echidna/Exercise-2.md
- Exercise-2.md
- Goal: check for correct access control of the token

First: try without the template!

Exercise 2 - Solution

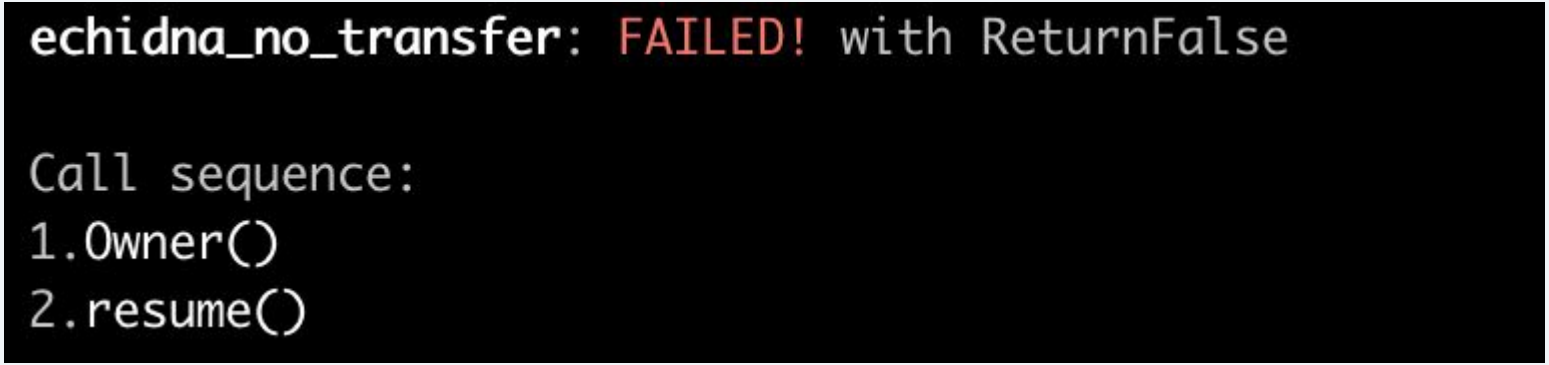
```
contract TestToken is Token {

   constructor() {
       paused();
       owner = 0x0; // lose ownership
   }

   // add the property
}
```

Exercise 2 - Solution

```
contract TestToken is Token {

   constructor() {
       paused();
       owner = 0x0; // lose ownership
   }

   function echidna_no_transfer() view returns(bool) {
       return is_paused == true;
   }
}
```

Exercise 2 - Solution

```
$ echidna-test solution.sol
```



```
echidna_no_transfer: FAILED! with ReturnFalse

Call sequence:
1.Owner()
2.resume()
```

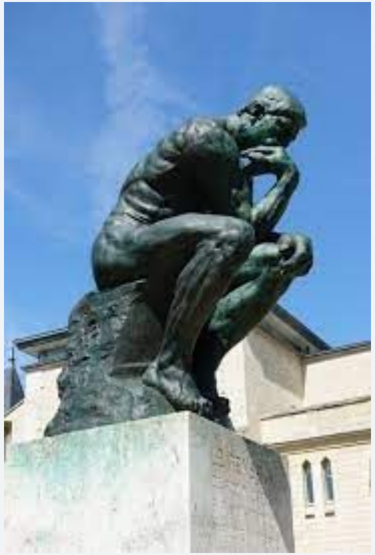
How to define good invariants

Defining good invariants

- **Start small, and iterate**
- **Steps**
  1. Define invariants in English
  2. Write the invariants in Solidity
  3. Run Echidna
     - If invariants broken: investigate
     - Once all the invariants pass, go back to (1)

Identify invariants

- **Sit down and think about what the contract is supposed to do**
- **Write the invariant in plain English**



Identify invariants: Maths

- **Math library**
    - Commutative property
        - 1 + 2 = 2 + 1
    - Identity property
        - 1 * 2 = 2
    - Inverse property
        - x + (-x) = 0

Identify invariants: tokens

- **ERC20.total_supply**
    - No user should have a balance > total_supply
- **ERC20.transfer:**
    - After calling `transfer`
        - My balance should have decreased by the amount
        - The receiver's balance should have increased by the amount
    - If the destination is myself, my balance should be the same
    - If I don't have enough funds, the transaction should revert/return false

Write invariants in Solidity

- **Identify the target of the invariant**
  - Function-level invariant
    - Ex: arithmetic’s associativity
    - Usually stateless invariants
    - Can craft scenario to test the invariant
  - System-level invariant
    - Ex: user’s balance < total supply
    - Usually stateful invariants
    - All functions must be considered

Function-level invariant

- **Inherit the targets**
- **Create function and call the targeted function**
- **Use assert to check the property**

```
contract TestMath is Math{
    function test_commutative(uint a, uint b) public {
        assert(add(a, b) == add(b, a));
   }
}
```

System level invariant

- **Require initialization**
    - Simple initialization: constructor
    - Complex initialization: leverage your unit tests framework with etheno
- **Echidna will explore all the other functions**

System level invariant

```
contract TestToken is Token {
    address echidna_caller =
0x00a329C0648769a73afAC7F9381e08fb43DBEA70;

    constructor() public{
        balances[echidna_caller] = 10000;
    }

    function test_balance() public{
        assert(balances[echidna_caller] <= 10000);
    }
}
```

Demo

Demo

```
/// @notice Allow users to buy token. 1 ether = 10 tokens
/// @param tokens The numbers of token to buy
/// @dev Users can send more ether than token to be bought, to give gifts to the
team.
function buy(uint tokens) public payable{
    _valid_buy(tokens, msg.value);
    _mint(msg.sender, tokens);
}


/// @notice Compute the amount of token to be minted. 1 ether = 10 tokens
/// @param desired_tokens The number of tokens to buy
/// @param wei_sent The ether value to be converted into token
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```

Demo

- **buy is stateful**
- **_valid_buy is stateless**
  - Start with it

Demo

- **What invariants?**

```
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```

Demo

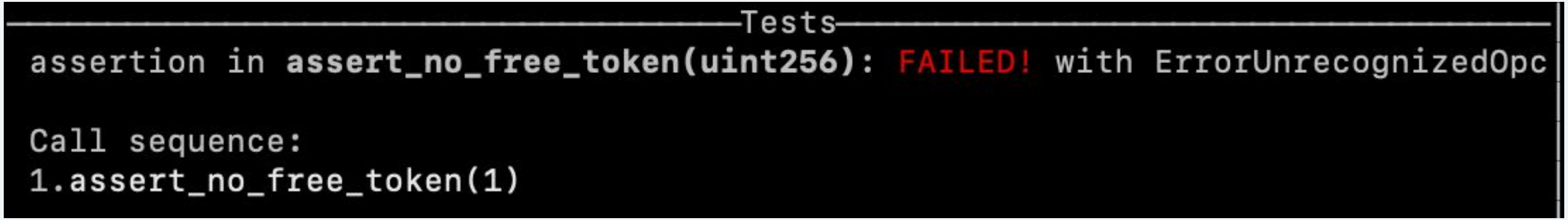
- **What invariants?**
    - If wei_sent is zero, desired_tokens must be zero

```
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```

Demo

```
function assert_no_free_token(uint desired_amount) public {
     require(desired_amount>0);
     _valid_buy(desired_amount, 0);
     assert(false); // this should never be reached
}
```

Demo



```
────────────────────────────────────Tests────────────────────────────────────
 assertion in assert_no_free_token(uint256): FAILED! with ErrorUnrecognizedOpc

 Call sequence:
 1.assert_no_free_token(1)
```

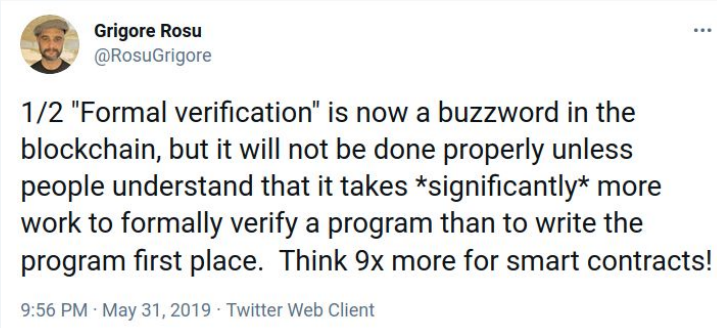
Comparison with similar tools

Other fuzzers

- **Inbuilt in dapp, brownie, foundry, ..**
- **Might be easier for simple test, however**
  - Less powerful
  - Require specific compilation framework

Formal methods based approach

- **Manticore, KEVM, Certora, ..**
- **Provide proofs, however**
  - More difficult to use
  - Return on investment is significantly higher with fuzzing



Grigore Rosu
@RosuGrigore

1/2 "Formal verification" is now a buzzword in the blockchain, but it will not be done properly unless people understand that it takes *significantly* more work to formally verify a program than to write the program first place. Think 9x more for smart contracts!

9:56 PM · May 31, 2019 · Twitter Web Client

Echidna’s advantages

- **Echidna has unique additional advanced features**
  - Can target high gas consumption functions
  - Differential fuzzing
  - Works with any compilation framework
  - Different APIs
    - Boolean property, assertion, dapptest/foundry mode, …
- **Free & open source**

Conclusion

Conclusion

- https://github.com/crytic/echidna
- To learn more: github.com/crytic/building-secure-contracts
- Start by writing invariants in English, then write Solidity properties
  - Start simple and iterate
- Your mission
  - Try Echidna on your current project

ToB is hiring (https://jobs.lever.co/trailofbits)

- Security Consultants & Apprentices
- The road to the apprenticeship blogpost

Additional slides

Where to focus?

Where to focus?

- **In practice: you don’t know where the bugs are**
- **Code coverage vs behavior coverage**
  - Cover as many functions as possible or;
  - Focus on specific components?

Where to focus?

- **Try different strategies**
    - Behavior coverage first
        - Focus on 1 or 2 components
    - Code coverage first
        - Cover many functions with simple properties
    - Alternate: 1 day on behavior coverage, then 1 day on code coverage, …
    - No right or wrong approach: try and see what works for you

Where to focus?

- **Start simple, then think about composition, related behaviors, etc...**
  - Can transfer and transferFrom be equivalent?
    - `transfer(to, value) ?= transferFrom(msg.sender, to, value)`
  - Is transfer additive-like?
    - `transfer(to, v0), transfer(to, v1) ?= transfer(to, v0 + v1)?`

Where to focus?

- **Start simple, then think about composition, related behaviors, etc…**
    - Can transfer and transferFrom be equivalent?
        - `transfer(to, value) ?= transferFrom(msg.sender, to, value)`
    - Is transfer additive-like?
        - `transfer(to, v0), transfer(to, v1) ?= transfer(to, v0 + v1)?`
        - Spoiler: this won't hold; why?

Where to focus?

- Building your own experience will make you more efficient over time
- Learn on how to think about invariants is a key component to write better code