TRAIL OF BITS

Introduction to fuzzing

Who am I?

- **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, Caracal, solc-select, ..

ToB Twitter list

Agenda

- How to find bugs?
- What is property based testing?
- How to define good invariants?

How to Find Bugs?

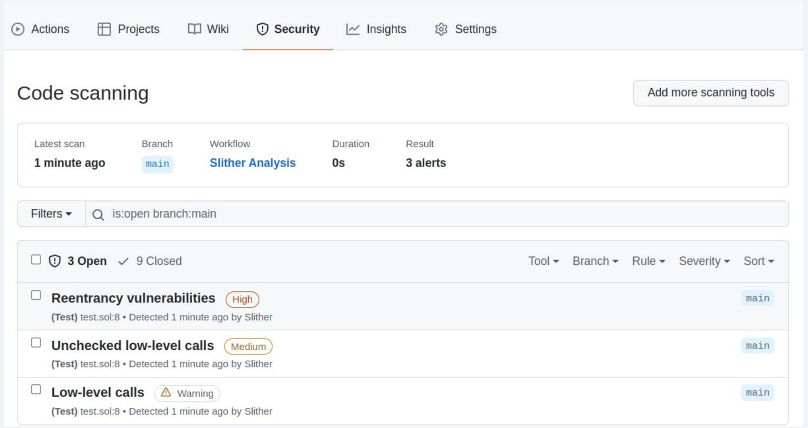
```
/// @notice Allow users to buy token. 1 ether = 10 tokens
/// @param tokens The numbers of token to buy
/// @dev Users can send more ether than token to be bought, to give gifts to the
team.
function buy(uint tokens) public payable{
    _valid_buy(tokens, msg.value);
    _mint(msg.sender, tokens);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


/// @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**

Fully automated analysis

- **Benefits**
  - Quick & easy to use
- **Limitations**
  - Cover only some class of bugs
- **Example: [Slither]**



Code scanning

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

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

https://github.com/crytic/slither-action

Semi automated analysis

- **Benefits**
    - Great for logic-related bugs
- **Limitations**
    - Require human in the loop
- **Example: Property based testing with Echidna**

What is property based testing?

Fuzzing

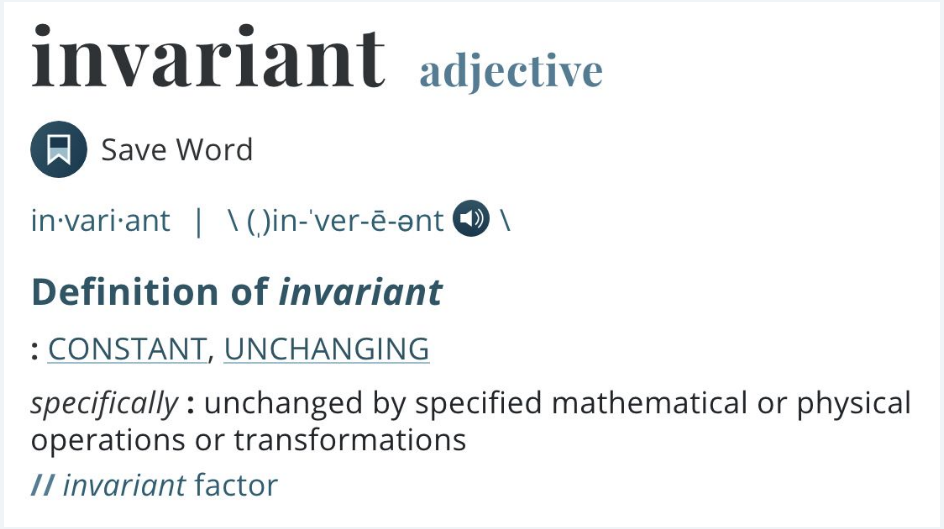
- **Stress the program with random inputs**
- **Fuzzing is well established in traditional software security**
  - AFL, Libfuzzer, go-fuzz, ..

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

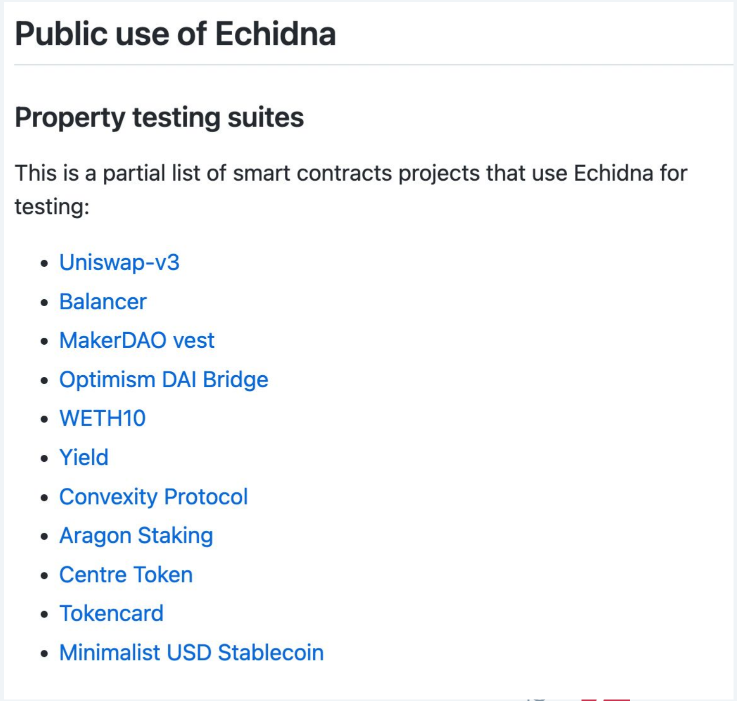
Invariant - Token’s total supply

User balance never exceeds total supply

Echidna

Echidna

- **Smart contract fuzzer**
- **Open source: github.com/crytic/echidna**
- **Heavily used in audits & mature codebases**
- **Focused in easy to use**
  - Solidity invariants
  - Github action
  - All compilation frameworks



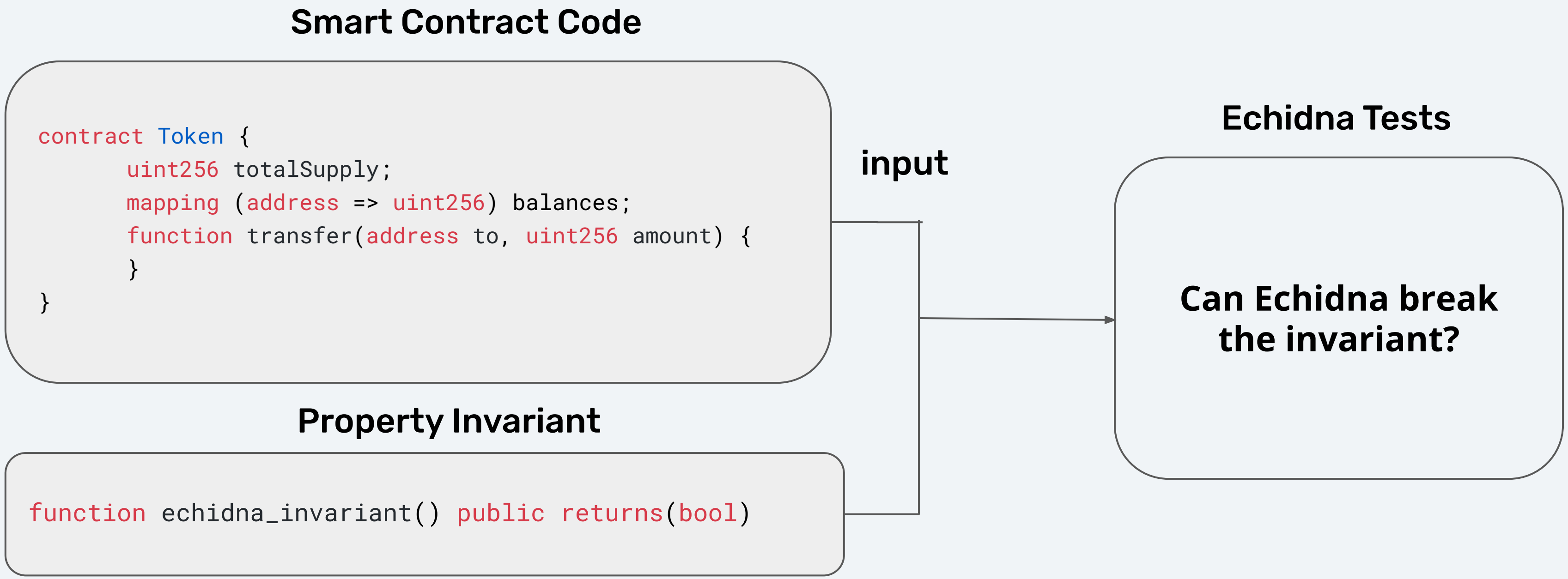
Public use of Echidna

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

Echidna - Overview



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

```
function echidna_invariant() public returns(bool)
```

Example - Token

```
contract Token is Ownable, Pausable {
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 value) public whenNotPaused {
        // unchecked to save gas
        unchecked {
            balances[msg.sender] -= value;
            balances[to] += value;
        }
    }
}
```

Example - User balance never exceeds total supply

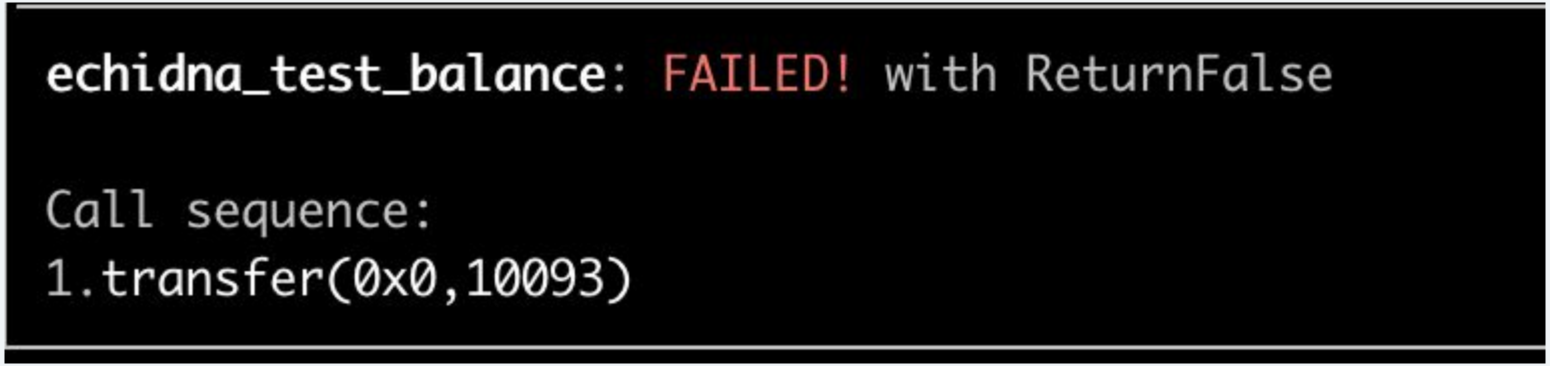
```
contract TestToken is Token {

   address echidna_caller = msg.sender;



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


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

Example - User balance never exceeds total supply

```
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 solution.sol
```



```
echidna_test_balance: FAILED! with ReturnFalse

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

Example - Token

```
contract Token is Ownable, Pausable {
    mapping(address => uint256) public balances;

    function transfer(address to, uint256 value) public whenNotPaused {
        // unchecked to save gas
        unchecked {
            balances[msg.sender] -= value;
            balances[to] += value;
        }
    }
}
```

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: 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

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**

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 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 specific initialization**
  - Constructors
- **Echidna will explore all the other functions**

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

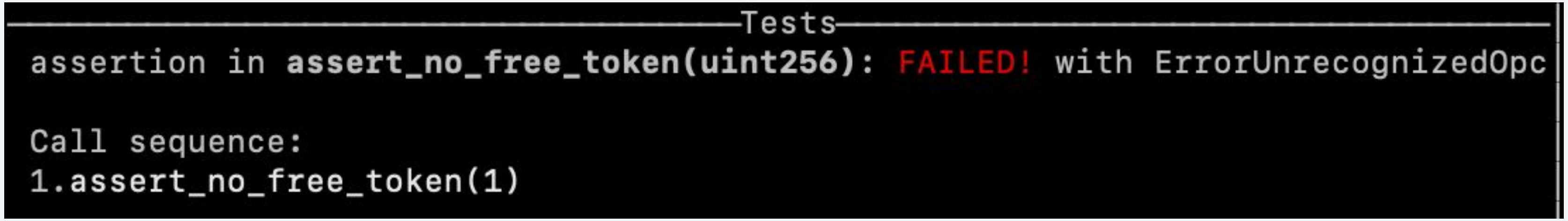
```
function _valid_buy(uint desired_tokens, uint wei_sent) internal view{
    uint required_wei_sent = (desired_tokens / 10) * decimals;
    require(wei_sent >= required_wei_sent);
}
```

Demo

- **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);
}

function assert_no_free_token(uint desired_amount) public {
     _valid_buy(desired_amount, 0);
     assert(desired_amount == 0);
}
```

Demo



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

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

Demo

```
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);
}
```

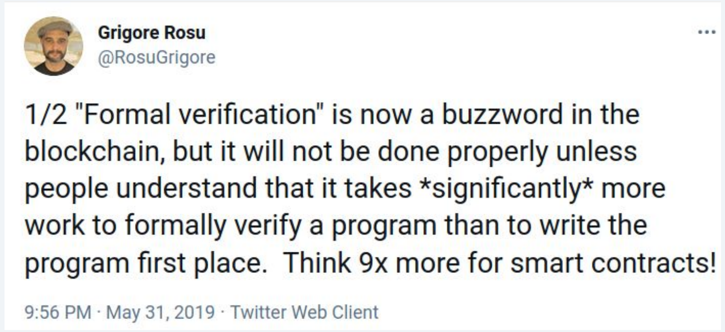
Comparison with similar tools

Other fuzzers

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

Formal methods based approach

- **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**

Medusa

- https://github.com/crytic/medusa
- Rewrite of Echidna in Go
- Still experimental, but we are looking for feedback

Conclusion

Conclusion

- **To learn more**
  - Secure-contracts.com
  - github.com/crytic/properties
- **Start with invariants in English, then Solidity**
  - Start simple and iterate
  - Try Echidna on your current project

Do you want help? Invariant as a service: