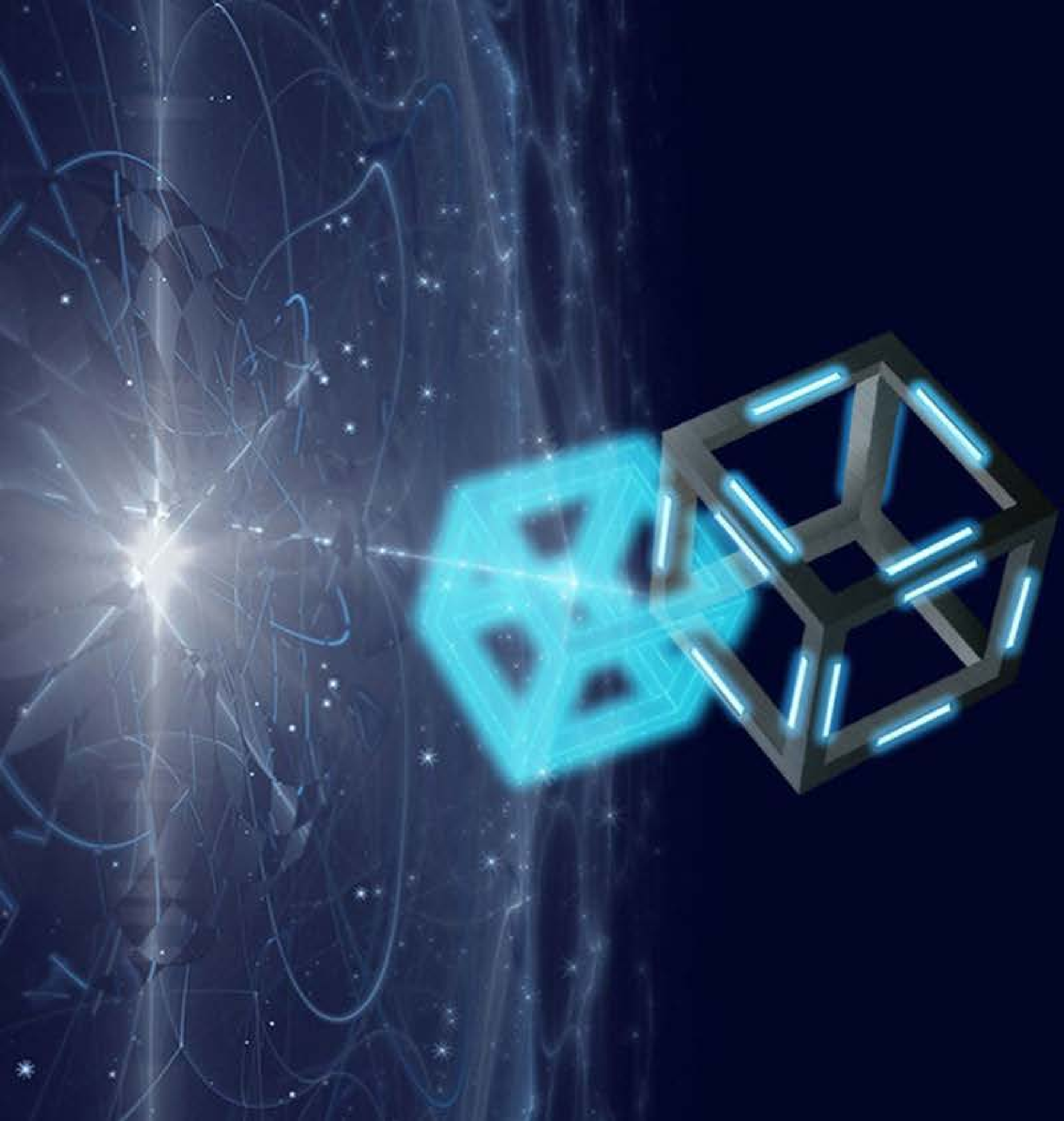
ETHEREUM SMART CONTRACT DEVELOPMENT

Fuzzing Smart Contracts With Echidna



Installing Echidna

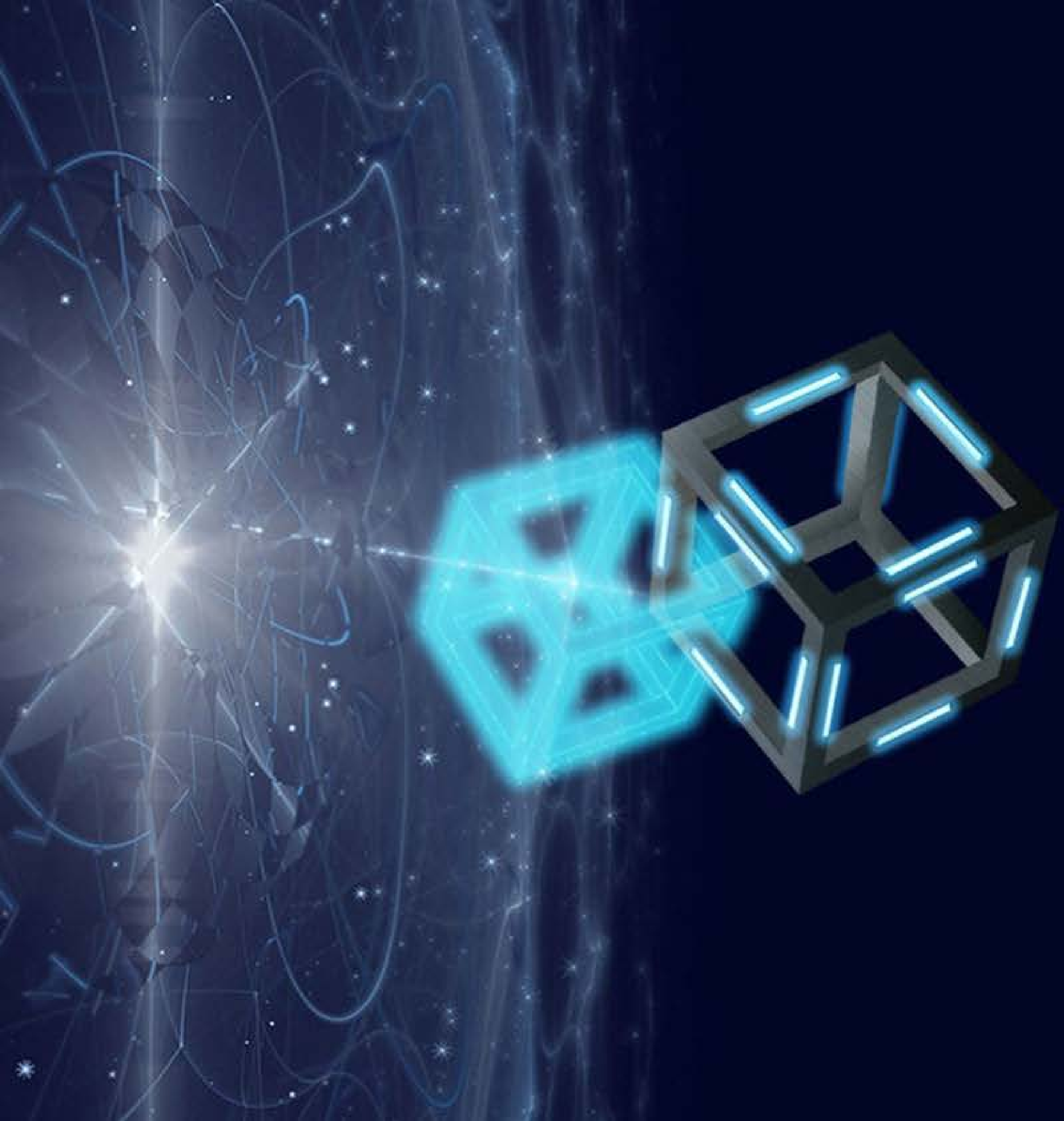
Installing Echidna

- **Prerequisites:**
  - Install Python 3.8+
    - Download Python: https://www.python.org/downloads/
    - Installation: https://www.datacamp.com/blog/how-to-install-python
  - Install Solc-Select (quickly switch between Solidity compiler versions)
    - Official Github link: https://github.com/crytic/solc-select
    - Installation: ***pip3 install solc-select***
    - **Using Solc-Select :**
      - Check the current solc version: ***solc --version***
      - Install a specific solc version: ***solc-select install 0.8.20***
      - Use a specific version: ***solc-select use 0.8.20***
  - Install Slither: ***pip3 install slither-analyzer***

Installing Echidna

- macOS & Linux (with Homebrew): ***brew install echidna***

- Windows (install standalone => add to C:\Windows - environment path is already set): https://github.com/crytic/echidna/releases



Echidna Overview

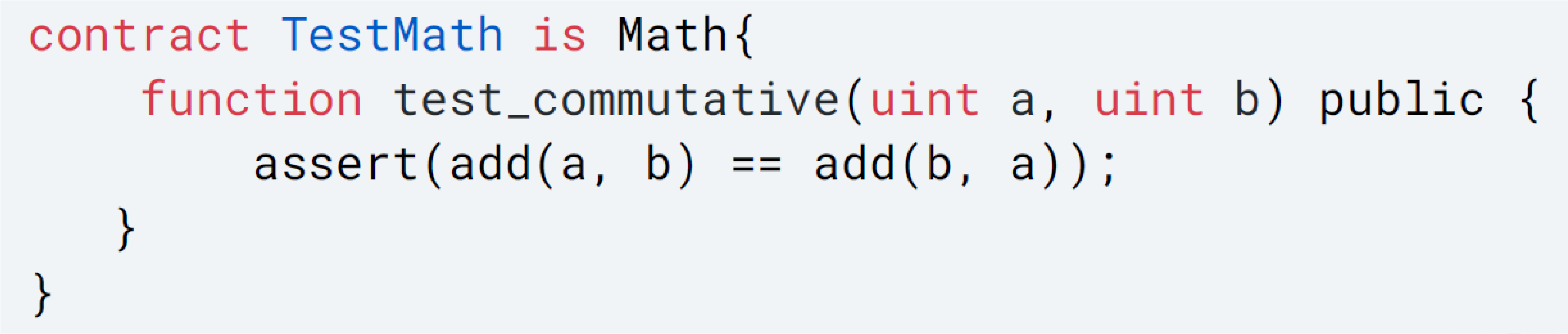
Echidna Overview

- Echidna is a property-based fuzzer that tries to break user-defined invariant
- Fuzzing involves generating more or less random inputs to find bugs in a program
- For each invariant, it generates random sequences of calls to the contract and checks if the invariant holds. If it can find some way to falsify the invariant, it prints the call sequence that does so
- Echidna provides two important testing modes: Property testing (default) and Assertion testing

Types of Invariants

Function-level invariants:

- Doesn’t rely on much of the system - mostly stateless
- Can be tested in an isolated fashion
- Inherit target contract, create test function(s) that calls the target function(s), use "assert" to check the property

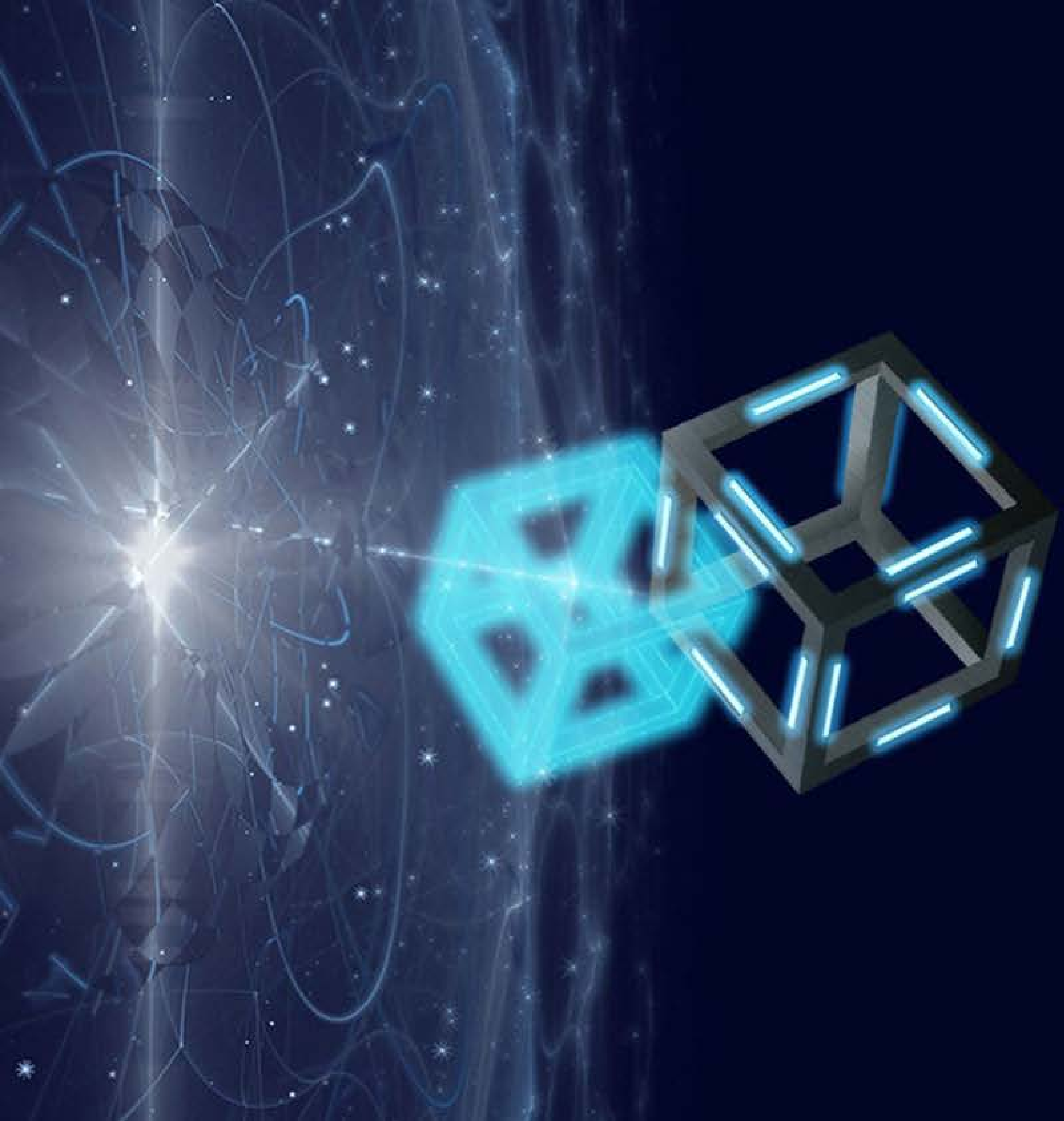


```
contract TestMath is Math{
    function test_commutative(uint a, uint b) public {
        assert(add(a, b) == add(b, a));
    }
}
```

Types of Invariants

System-level invariants:

- Relies on the deployment of a large part or the entire system
- Invariants are usually stateful
- Requires deployment/initialization:
  - deploy/initialize everything in the constructor
  - for large systems, a framework like ***Etheno*** can be used: https://github.com/crytic/etheno



Echidna
Testing Modes

Property Testing Mode

- By default, Echidna uses the "property" testing mode, which reports failures using special functions called properties
- Testing functions are named with the prefix: ***echidna_***
- Testing functions take no arguments and always return a boolean value
- Any side effect (changing a storage variable) will be reverted at the end of the execution of the property
- Properties pass if they return true and fail if they return false or revert
- Alternatively, properties that start with "***echidna_revert_***" will fail if they return any value (true or false) and pass if they revert

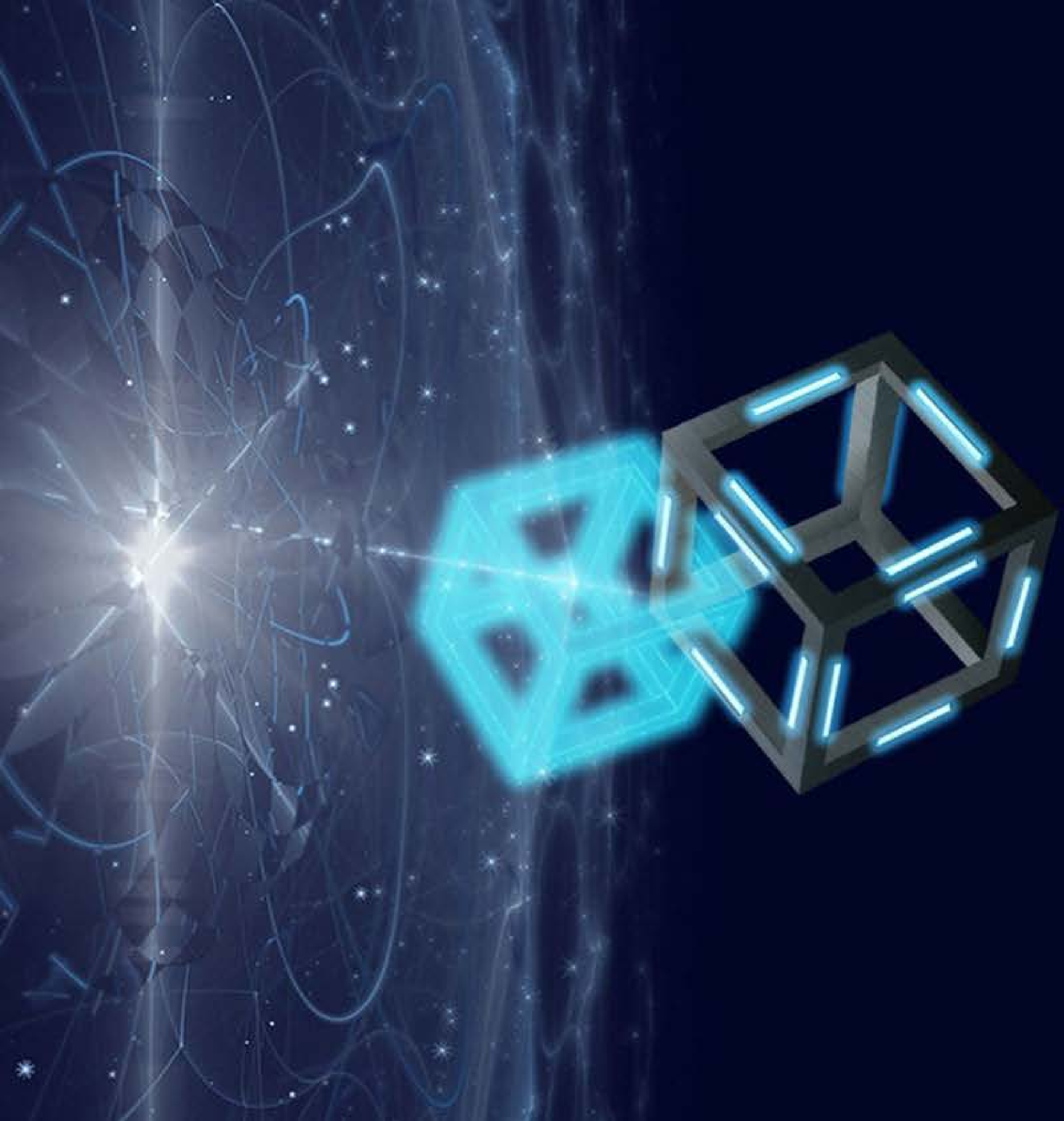
Example property (invariant): balance should never go below 20

```
function echidna_check_balance() public returns (bool) {
        return(balance >= 20);
}
```

Assertion Testing Mode

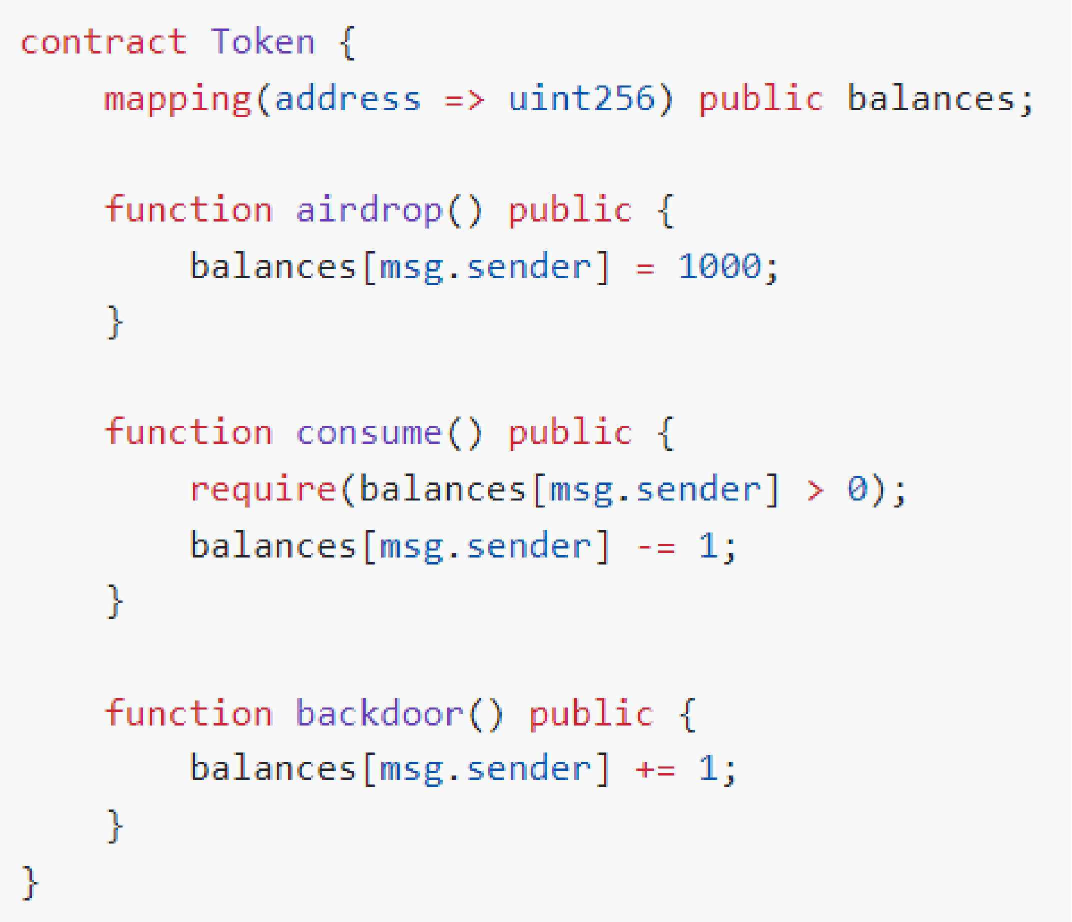
- This mode is well suited for more complex code that requires checking or changing state variables
- Test functions do not require any particular name
- Any number of arguments is allowed in the test function
- Side effects are retained if they do not revert
- The following needs to be added to the config.yaml file: ***testMode: assertion***

- The following statements trigger a failure:
  - assert(false);
  - emit AssertionFailed(...);



Property Test Mode Example

Example: Property Testing Mode



```
contract Token {
    mapping(address => uint256) public balances;

    function airdrop() public {
        balances[msg.sender] = 1000;
    }

    function consume() public {
        require(balances[msg.sender] > 0);
        balances[msg.sender] -= 1;
    }

    function backdoor() public {
        balances[msg.sender] += 1;
    }
}
```

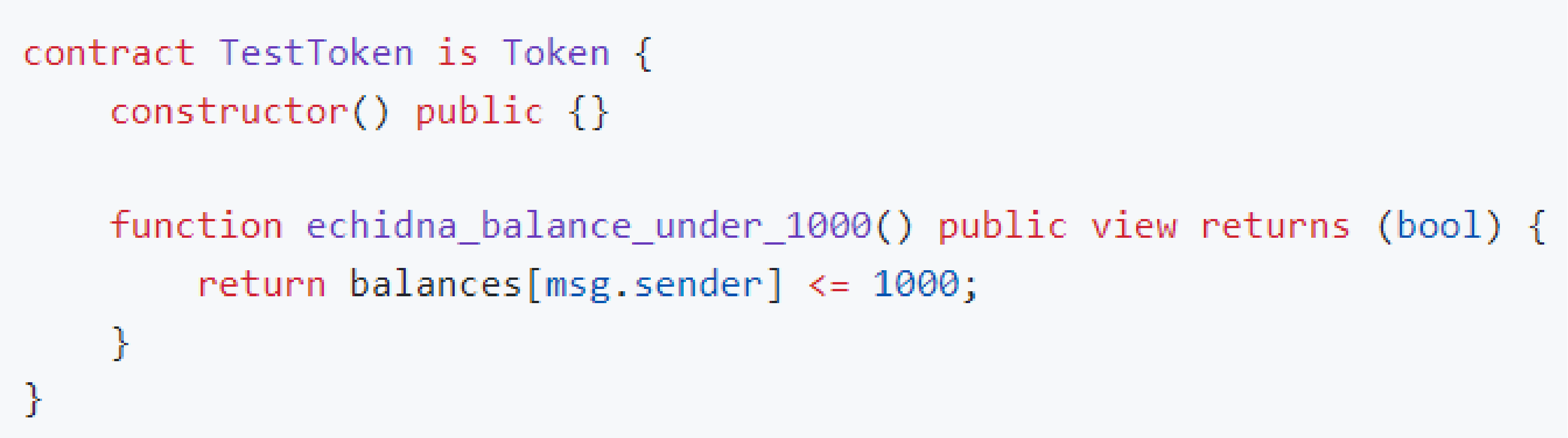
- **Required contract property:**

  Anyone can hold a maximum of 1000 tokens

- **Echidna will:**
  - Automatically generate arbitrary transactions to test the property
  - Report any transactions that lead a property to return false or throw an error

Example: Property Testing Mode

- The following property checks that the caller can have no more than 1000 tokens:



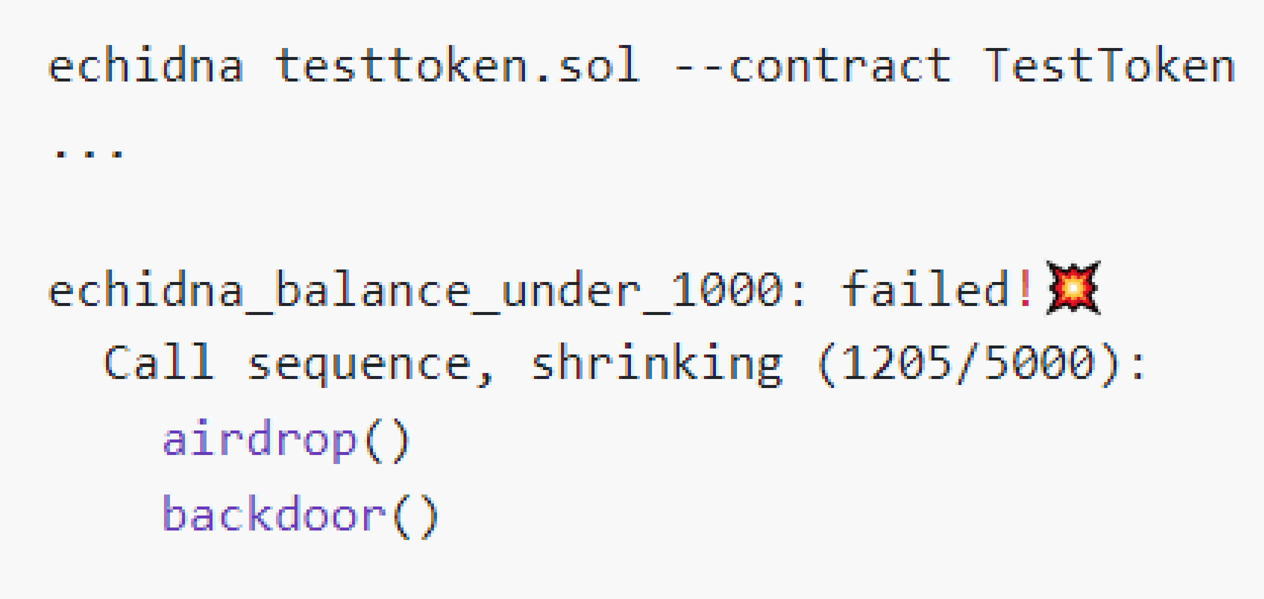
```
contract TestToken is Token {
    constructor() public {}

    function echidna_balance_under_1000() public view returns (bool) {
        return balances[msg.sender] <= 1000;
    }
}
```

- If a contract needs specific initialization, it should be done in the constructor
- There are some specific addresses in Echidna:
  - 0x30000 calls the constructor
  - 0x10000, 0x20000, and 0x30000 randomly call other functions

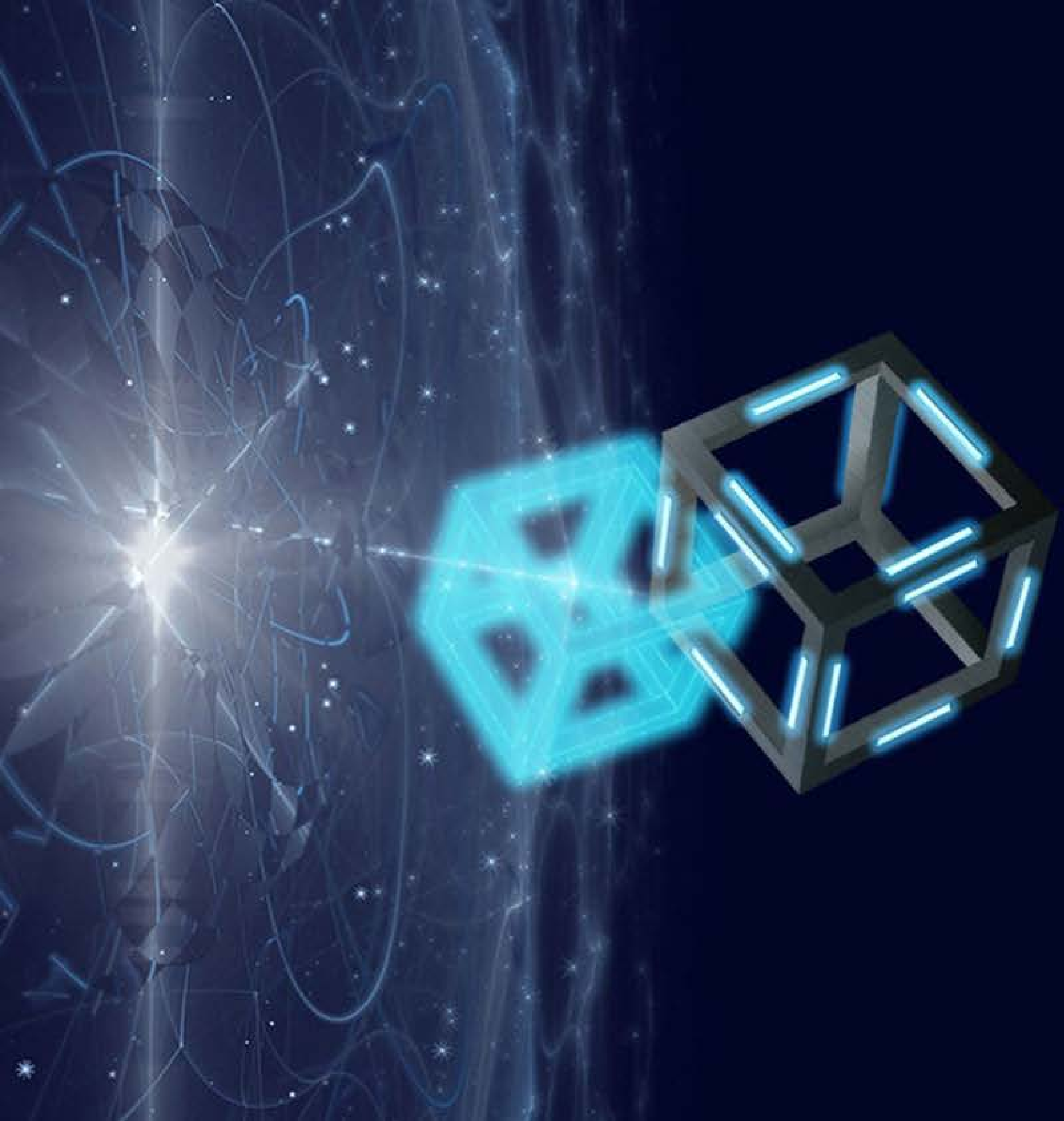
Running Echidna

- echidna contract.sol
- echidna contract.sol --contract myContract
- echidna contract.sol --contract myContract --config config.yaml



```
echidna testtoken.sol --contract TestToken
...

echidna_balance_under_1000: failed!💥
  Call sequence, shrinking (1205/5000):
    airdrop()
    backdoor()
```



Optimization, Configuration, Coverage...

Optimizing the Fuzzer Performance

- Each time a "require" fails, computation is wasted => should be prevented to maximize each transaction sequence

- Bound inputs with strong pre-conditions or arithmetic manipulation:

  - Pre-condition: ***require(usdc.balanceOf(msg.sender) > 0))***

  - Arithmetic manipulation: ***if (abs(x) == abs(y)) { y = x + 1 };***

  - Modular arithmetic: ***uint256 x = inputValue % UPPER_BOUND;*** => bound to [0, UPPER_BOUND - 1]

Structuring Tests

Pre-condition checks:

- Barriers of entry for the fuzzer
- Don't test this property unless these pre-conditions are true => for example: ***require(usdc.balanceOf(msg.sender) > 0))***

Action:

- Execute function/scenario to be tested => for example: ***usdc.transfer(to, amount)***

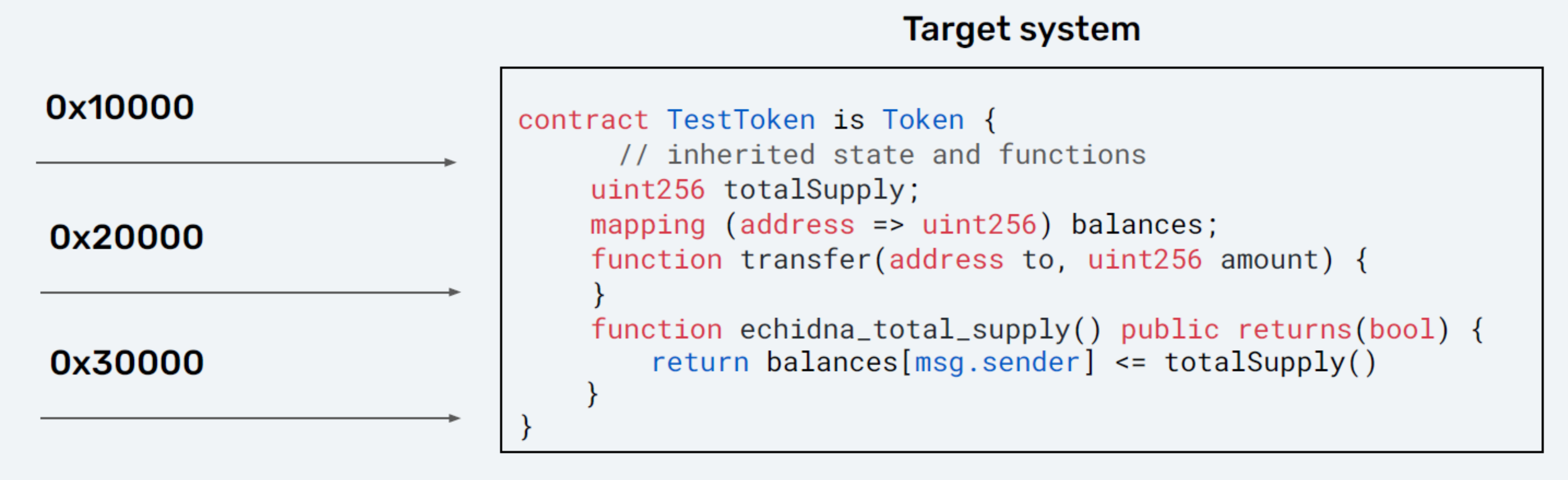
Post-condition checks:

- These are the "truths" we are testing
- Must test both happy and not-so-happy path (try/catch)
- Example: ***assert(usdc.balanceOf(msg.sender) < 1000);***

Internal versus External Testing

Internal testing:

- ➢ Use of inheritance to test the target contract
- ➢ Easy to set up
- ➢ Get the state and all public/external functions of the inherited contracts
- ➢ msg.sender is preserved
- ➢ Not suited for complex systems

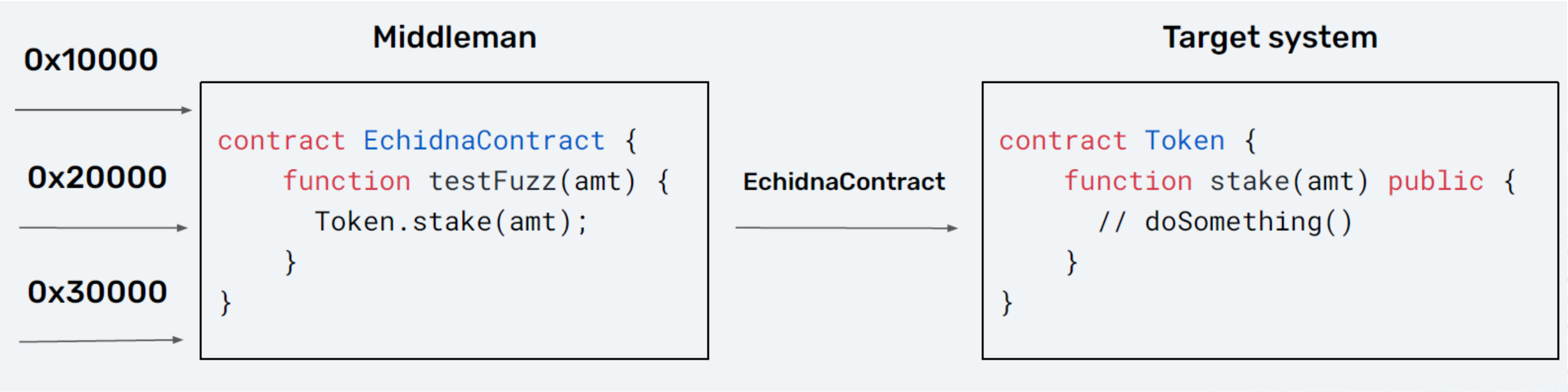


```
contract TestToken is Token {
      // inherited state and functions
    uint256 totalSupply;
    mapping (address => uint256) balances;
    function transfer(address to, uint256 amount) {
    }
    function echidna_total_supply() public returns(bool) {
        return balances[msg.sender] <= totalSupply()
    }
}
```

Internal versus External Testing

External testing:

- Use of external calls to the target contract(s)
- More difficult to set up
- msg.sender is not preserved
- Used for complex systems



```
contract EchidnaContract {
    function testFuzz(amt) {
      Token.stake(amt);
    }
}
```

```
contract Token {
    function stake(amt) public {
      // doSomething()
    }
}
```

Stateless versus Stateful Testing

- Both testing modes can be used, in either stateful (default) or stateless mode (using seqLen: 1 in config.yaml)

- In stateful mode, Echidna will maintain the state between each function call and attempt to break the invariants. In stateless mode, Echidna will discard state changes during fuzzing

Specific addresses in Echidna:

- 0x30000 calls the constructor
- 0x10000, 0x20000, and 0x30000 randomly call other functions

Echidna Configuration

- Default yaml: https://github.com/crytic/echidna/blob/master/tests/solidity/basic/default.yaml
- Config options: https://github.com/crytic/echidna/wiki/Config

- Run Echidna: echidna contract.sol --contract myContract --config config.yaml

Blacklisting functions:

```
filterBlacklist: true
filterFunctions: ["myContract.function1(uint256,uint256)", " myContract.function1()"]
```

Whitelisting functions:

```
filterBlacklist: false
filterFunctions: ["myContract.function3()", " myContract.function4()"]
```

Measure Gas Consumption:

```
estimateGas: true
```

Echidna Coverage

- Coverage is the tracking of what code was touched by the fuzzer. Echidna can save coverage information in a directory specified with the corpusDir config option in the config.yaml file
- Create a corpus directory in the project root - e.g.: corpus/
- Add to the config.yaml: ***corpusDir: "corpus"***
- Run Echidna with the config.yaml: ***echidna myContract.sol --config config.yaml***
- Check the ***corpus/covered.*.txt*** file - example:

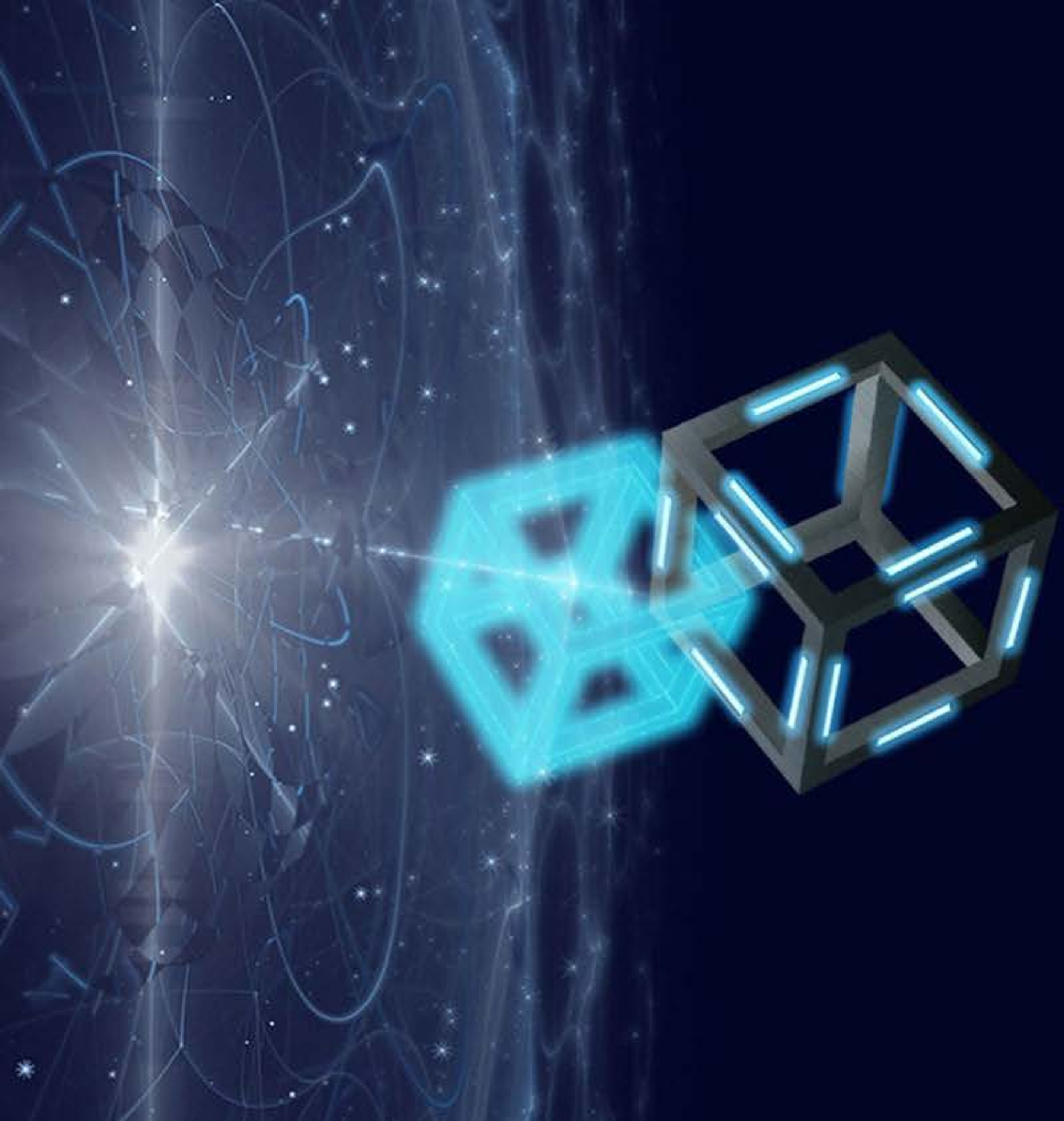
```
*r    function set0(int val) public returns (bool){
*       if (val % 100 == 0)
*         flag0 = false;
```

* : if an execution ended with a STOP - line was executed with no error

r : if an execution ended with a REVERT

o : if an execution ended with an out-of-gas error - common with loops

e : if an execution ended with any other error (zero division, assertion failure, etc)



Assertion Test Mode Example

Example: Assertion Test Mode - UniswapV2 Pool Liquidity

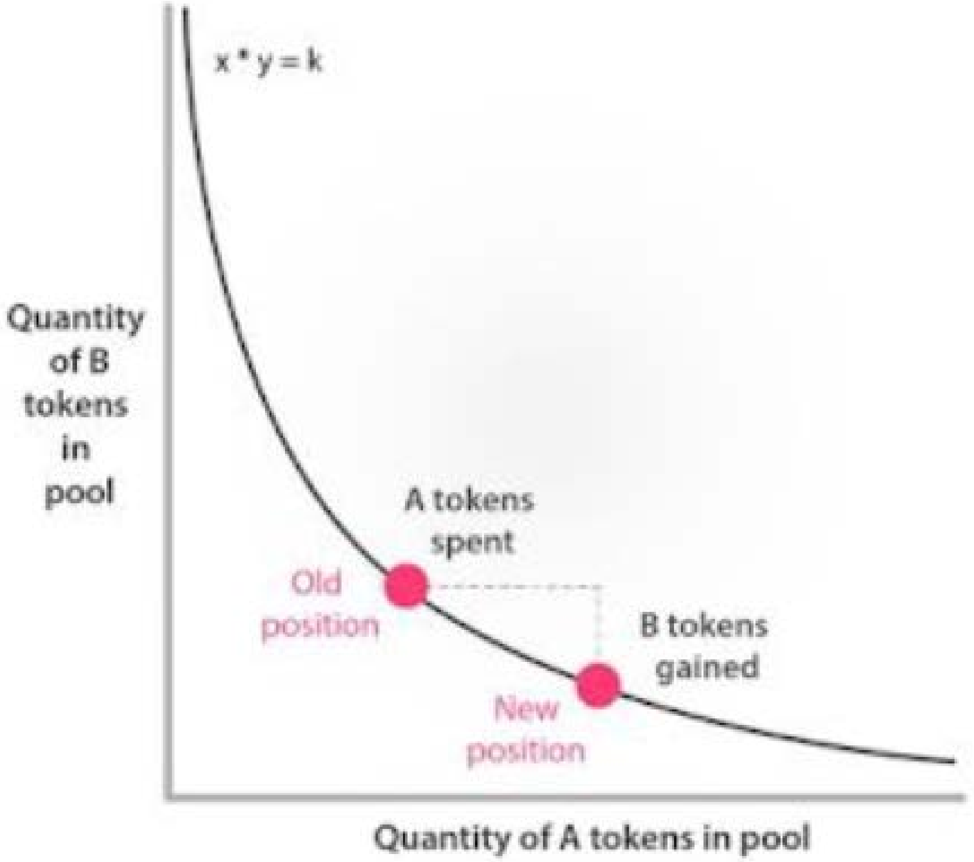
What is an Automated Market Maker (AMM):

- Exchange without orderbook
- Pricing is based on pool’s liquidity formula => $x * y = k$
- Price is calculated as ratio between two assets
- k (pool invariant) is constant

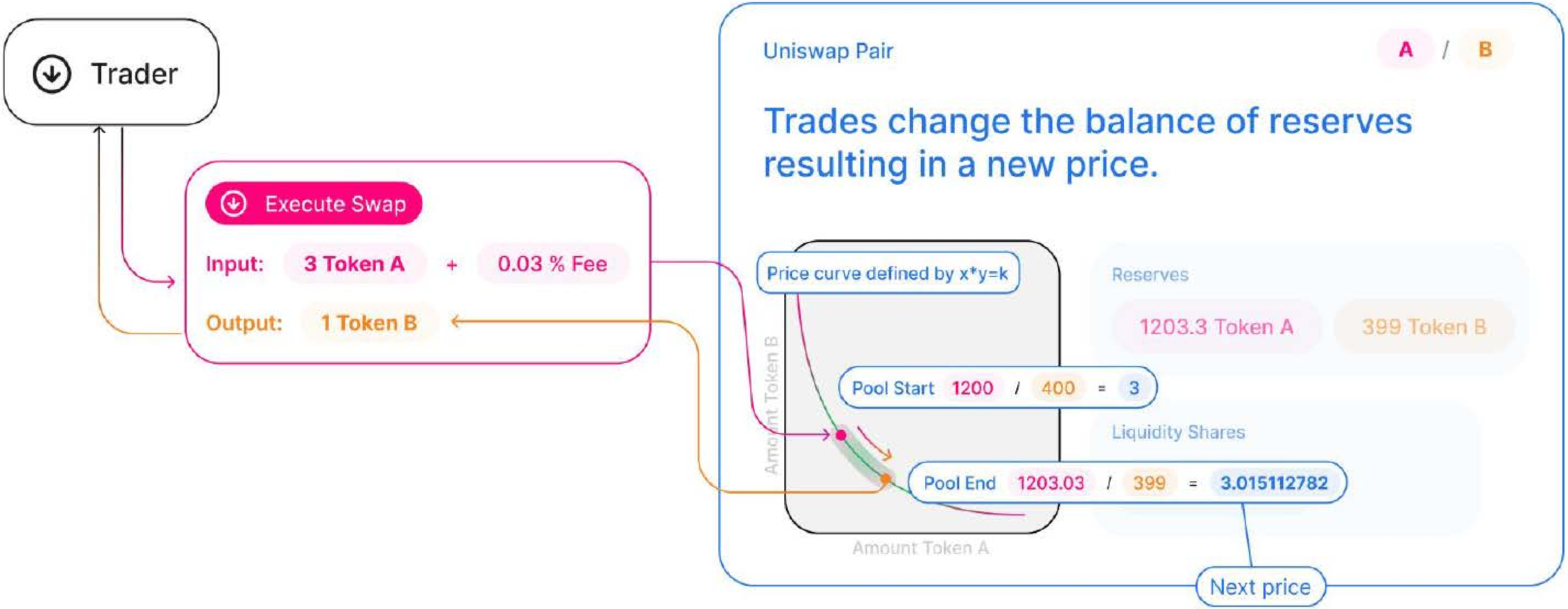
Example: Assertion Test Mode - UniswapV2 Pool Liquidity

Swap x amount of tokenA for tokenB - how much of tokenB do we get out?

- Δx: amount of tokenA we are swapping for tokenB (Δy)
- x * y = k
- (x + Δx) * (y - Δy) = k
  => the pool gains Δx and loses Δy
- Δy = y - (k / (x + Δx))



Example: Assertion Test Mode - UniswapV2 Pool Liquidity



Trades change the balance of reserves resulting in a new price.

Example: Assertion Test Mode - UniswapV2 Pool Liquidity

UniswapV2 Core has two important contracts:

- Factory: creates unique pair contracts for each pool
- Pair: represents liquidity pool, keeps track of token balances and contains the basic swapping logic

Invariants:

- Invariant 1: when we add liquidity to the pool, our stack of LP tokens needs to increase
- Invariant 2: when we add liquidity (x and y) to the pool, k = x * y needs to increase