TRAIL OF BITS

Echidna Workshop: Part 2/6 - Fuzzing Arithmetics and Functions

Before starting

- git clone https://github.com/crytic/echidna-streaming-series
- Open the **part2/** folder

Watch Part 1 here:
https://www.youtube.com/watch?v=bhb_y80iF8w

Upcoming workshops

Beginner

- Part 1: The Basics (Nov 16, 2022)
- **Part 2: Fuzzing arithmetics and functions (today)**

Intermediate

- Part 3: Introduction to AMM's invariants (Nov 30, 2022)
- Part 4: AMM fuzzing (Dec 6, 2022)

Advanced

- Part 5: Introduction to advanced DeFi's invariants (Dec 14, 2022)
- Part 6: Advanced DeFi invariants (Dec 21, 2022)

Who am I?

Anish Naik, Security Engineer I

(@anishrnaik)

Who You Should Follow

- **Troy Sargent (@0xalpharush)**
- **Josselin Feist (@montyly)**
- **Nat Chin (@0xicingdeath)**
- **Justin Jacob (@technovision99)**

Who are we?

Trail of Bits (@trailofbits)

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

Today’s agenda

- **Section 1**
  - Recap
  - What is ABDKMath?
  - Function-level properties of ABDKMath
  - Understanding precision loss
- **Section 2**
  - External testing vs. internal testing
  - Debugging with coverage reports
- **Conclusion**

Recap

Recap from “Part 1 - The Basics”

- **Property-based testing aids in identifying logical flaws in a codebase**
- **Echidna is used for property-based testing!**
- **Properties can be thought of as “truthy” values - or just boolean statements**
- **You can have:**
  - Function-level properties
  - System-level properties

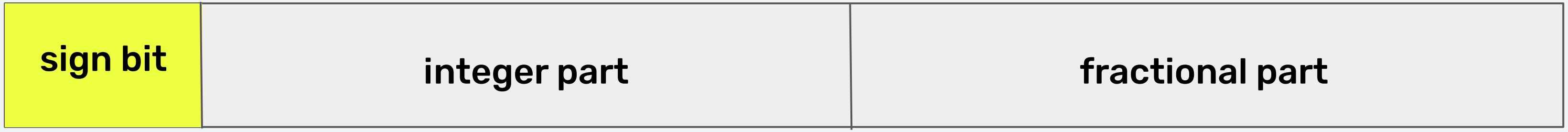
Recap from “Part 1 - The Basics”

- **Property-based testing aids in identifying logical flaws in a codebase**
- **Echidna is used for property-based testing!**
- **Properties can be thought of as “truthy” values - or just boolean statements**
- **You can have:**
  - **Function-level properties**
  - System-level properties

What is ABDKMath?

ABDKMath is a library for fixed-point arithmetic

- **The ABDKMath64x64 library implements fixed-point integer representation for rational numbers.**
- **Has basic operations like add() / sub() and more complex operations like inv() / log2()**



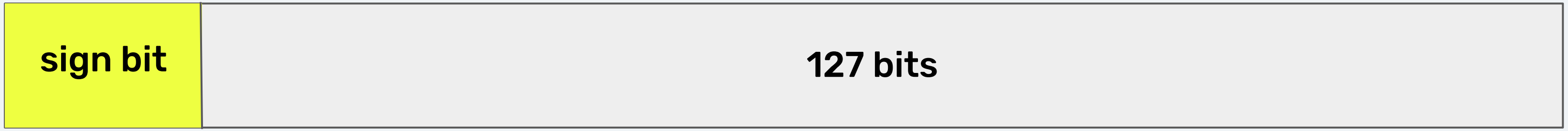
sign bit

integer part

fractional part

Representing a fixed-point value in Solidity

- **The 64.64 integer is represented in an int128**
- **Treat a 64.64 integer as a fraction**
  - Numerator is an int128, **variable**
  - Denominator is 2^64, **constant**
- **Since denominator stays the same, the int128 simply holds the numerator of the 64.64 integer**

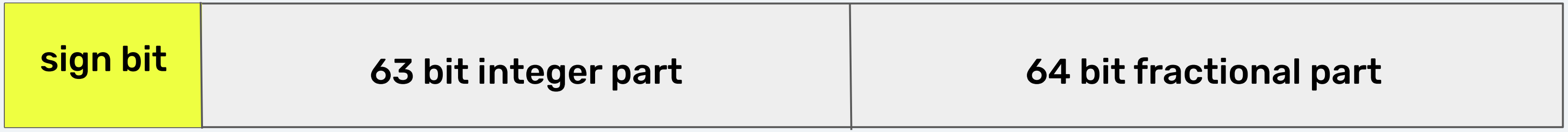


sign bit

127 bits

Representing a fixed-point value in Solidity

- **The 64.64 integer is represented in an int128**
- **Treat a 64.64 integer as a fraction**
    - Numerator is an int128, **variable**
    - Denominator is 2^64, **constant**
- **Since denominator stays the same, the int128 simply holds the numerator of the 64.64 integer**



sign bit	**63 bit integer part**	**64 bit fractional part**

Function-level properties of ABDKMath

Can test properties of addition and subtraction

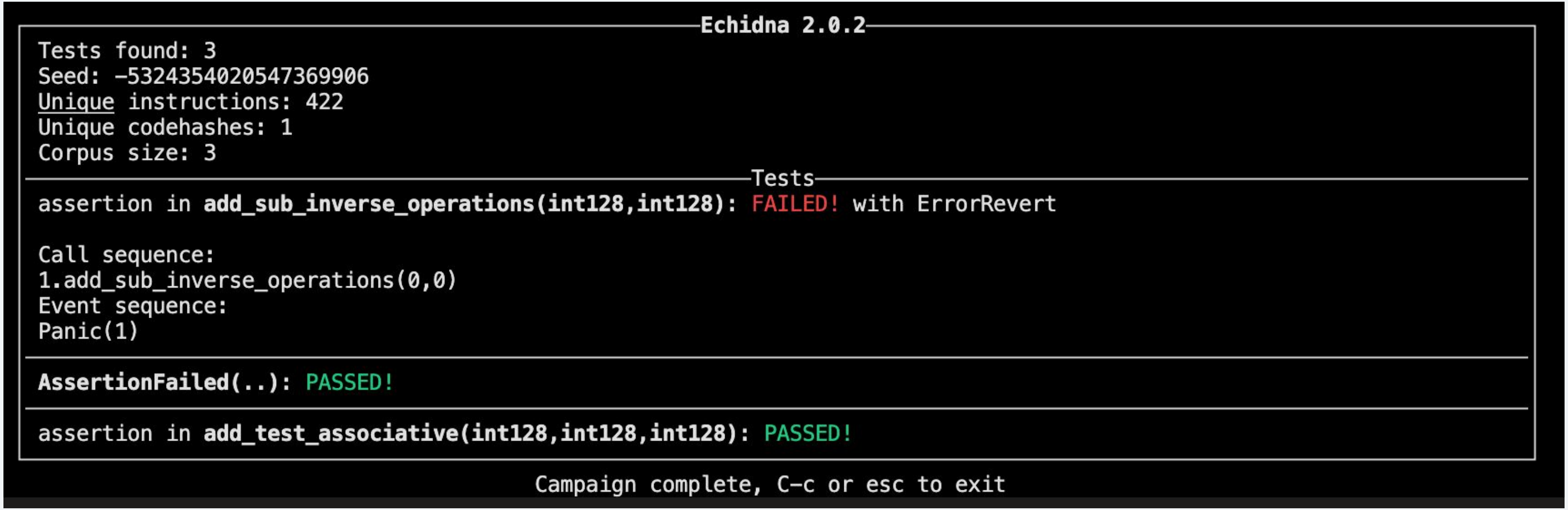
- **We can test simple mathematical properties of addition and subtraction**
  - Associative property of addition
    - (x + y) + z = x + (y + z)
  - Commutative property of addition
    - x + y = y + x
  - Addition and subtraction are inverse operations
    - (x + y) - y = x
  - Subtraction is NOT commutative
    - x - y != y - x

Can test properties of addition and subtraction

- **We can test simple mathematical properties of addition and subtraction**
  - **Associative property of addition**
    - (x + y) + z = x + (y + z)
  - Commutative property of addition
    - x + y = y + x
  - **Addition and subtraction are inverse operations**
    - (x + y) - y = x
  - Subtraction is NOT commutative
    - x - y != y - x

Let’s write some properties!

Why does add_sub_inverse_operations fail?

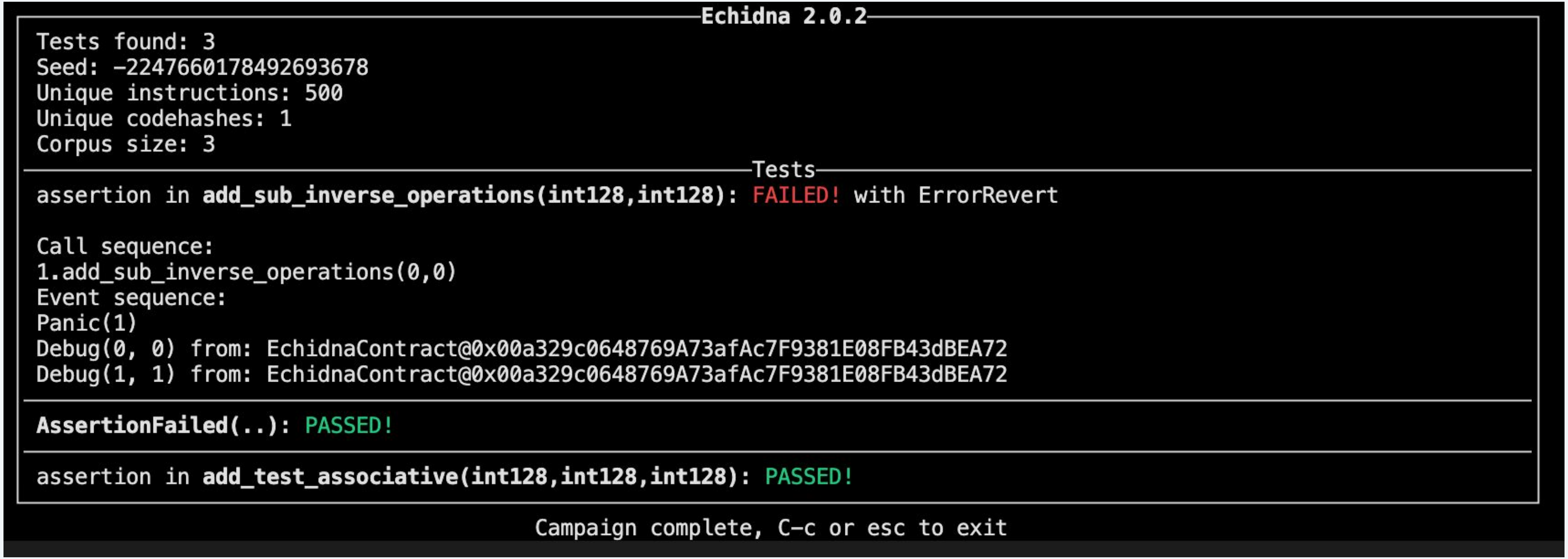


```
Echidna 2.0.2
Tests found: 3
Seed: -5324354020547369906
Unique instructions: 422
Unique codehashes: 1
Corpus size: 3
Tests
assertion in add_sub_inverse_operations(int128,int128): FAILED! with ErrorRevert

Call sequence:
1.add_sub_inverse_operations(0,0)
Event sequence:
Panic(1)

AssertionFailed(..): PASSED!

assertion in add_test_associative(int128,int128,int128): PASSED!

Campaign complete, C-c or esc to exit
```

Debugging failed tests with events

- **Emitting events aids in identifying the values that cause a property to fail**
- **Events are emitted *only* on the failing case**

Debugging failed tests with events



```
Echidna 2.0.2
Tests found: 3
Seed: -2247660178492693678
Unique instructions: 500
Unique codehashes: 1
Corpus size: 3
Tests
assertion in add_sub_inverse_operations(int128,int128): FAILED! with ErrorRevert

Call sequence:
1.add_sub_inverse_operations(0,0)
Event sequence:
Panic(1)
Debug(0, 0) from: EchidnaContract@0x00a329c0648769A73afAc7F9381E08FB43dBEA72
Debug(1, 1) from: EchidnaContract@0x00a329c0648769A73afAc7F9381E08FB43dBEA72

AssertionFailed(..): PASSED!

assertion in add_test_associative(int128,int128,int128): PASSED!

Campaign complete, C-c or esc to exit
```

Can test properties of multiplication and division

- **We can test simple mathematical properties of multiplication and division**
  - Associative property of multiplication
    - (x * y) * z = x * (y * z)
  - Commutative property of multiplication
    - x * y = y * x
  - Multiplication and division are inverse operations
    - (x * y) / y = x
  - Division is NOT commutative
    - x / y != y / x
  - Inverse of x is equal to 1 divided by x
    - inv(x) == 1/x

Can test properties of multiplication and division

- **We can test simple mathematical properties of multiplication and division**
    - Associative property of multiplication
        - (x * y) * z = x * (y * z)
    - Commutative property of multiplication
        - x * y = y * x
    - Multiplication and division are inverse operations
        - (x * y) / y = x
    - **Division is NOT commutative**
        - x / y != y / x
    - Inverse of x is equal to 1 divided by x
        - inv(x) == 1/x

Let’s write some more properties!

Structuring a function-level invariant

- **Pre-condition checks**
  - Barriers of entry for the fuzzer
  - “Don’t test this property unless these pre-conditions are true”
  - Example: `require(usdc.balanceOf(msg.sender) > 0))`
- **Action**
  - What you are testing
  - Example: `usdc.transfer(to, amount)`
- **Post-condition checks**
  - These are the “truths” you are testing
  - MUST test both happy and not-so-happy path (try/catch)
  - Example: `usdc.balanceOf(msg.sender)` has decreased by `amount`

Optimizing fuzzer performance

- **Goal: Want to maximize the value of each transaction sequence that Echidna runs**
  - Each time a `require` fails or under/overflow occurs, that's wasted computation
- **Solution: Bound your 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)`

Key takeaways

- **Do not test arithmetic operations by re-doing the operations**
  - Use *properties* of the arithmetic operations
  - Differential fuzz testing (assembly vs. non-assembly versions)
- **Emitting events can aid in debugging**
- **Formatting a function-level property test**
  - Pre-conditions: Scope the input space
  - Action: What we are testing
  - Post-conditions: The “truths” after the action
- **Optimize your fuzzer runs with pre-conditions and arithmetic manipulation**

Understanding precision loss

Defining precision loss

- **Precision loss occurs when the value of an operation cannot be represented in the underlying data type**
- **Addition and subtraction are vulnerable to overflows**
- **Multiplication / division / sqrt can all have variable amounts of precision loss**

Quantifying precision-loss

- **Various operations have different amounts of precision loss**
    - Multiplication and division: function of the magnitude of the two values that are being operated on
    - inv(x): logarithmic to the value of x
    - sqrt(x): precision-loss is half the number of bits in x
- **Helper functions are provided in Solution.sol to quantify precision-loss**
    - **equal_within_precision()**
    - **equal_within_tolerance()**
    - significant_digits_lost_in_mult()
    - equal_most_significant_bits_within_precision()

Resources for understanding precision loss

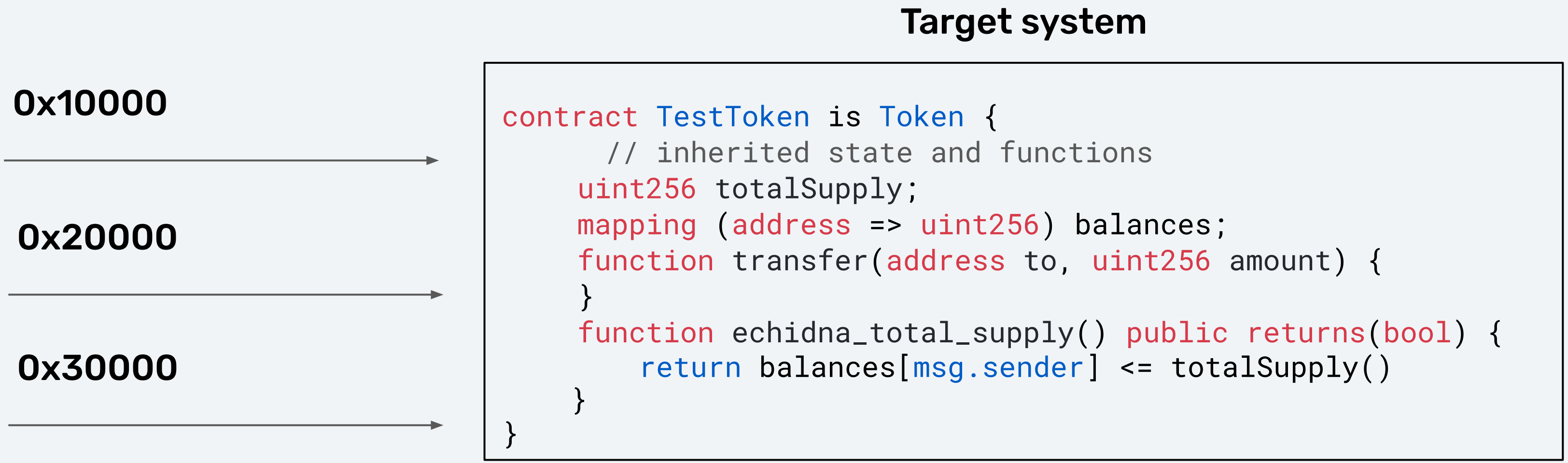
- A fixed-point introduction by example
- Binary Integer Fixed-point Numbers
- Fixed Point exponentiation
- On the Implementation of Fixed-point Exponential Function for Machine Learning and Signal Processing Accelerators
- Fixed Point Math library for MSP Processors - Texas Instruments

External testing vs internal testing

What is internal testing?

- **Internal testing is the use of *inheritance* to test the target contract**
- **Pros:**
  - Easy to set up
  - Get the state and all public/external functions of the inherited contracts
  - msg.sender is preserved
- **Cons:**
  - Not good for complex systems
  - Mostly viable for single-entrypoint systems

Visualizing internal testing



Target system

0x10000

0x20000

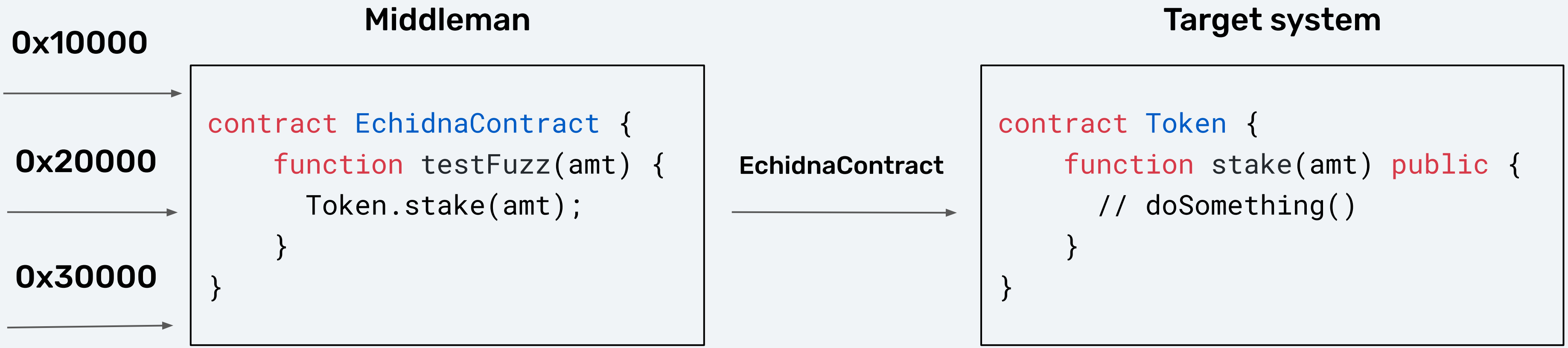
0x30000

```
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()
    }
}
```

What is external testing?

- **External testing is the use of *external* calls to the target system**
- **Pros:**
  - Good for complex systems with complex initialization
  - Good for multi-entrypoint systems
  - Mostly used in practice
- **Cons:**
  - Difficult to set up
  - msg.sender is *not* preserved

Visualizing external testing



0x10000

0x20000

0x30000

Middleman

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

EchidnaContract

Target system

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

Debugging with coverage reports

What is coverage?

Coverage is the tracking of what code was “touched” by the fuzzer (applies to any general testing utility).

How does Echidna report its coverage?

- **Echidna provides *coverage reports* to show what parts of the code were touched**
- **The report is a .txt file with all the target source code**
  - Each line in the report is marked with a(n):
    - *
    - r
    - o
    - e
    - None of the above = line was not covered

How to read a coverage report?

- ***: Execution ended with a STOP**
    - At some point, this line was executed with no errors
- **r: Execution ended with a REVERT**
    - At some point, this line caused the transaction to revert
- **o: Out-of-gas error**
    - Common with loops
- **e: Execution ended with any other error**
    - E.g. zero division

So why do I care about coverage reports?

- **It is a crucial debugging tool that will greatly improve your testing efforts**
- Provides a **guarantee** that your tests ran as expected.
- You should *not* run Echidna without coverage enabled
  - CLI: `--corpus-dir <name-of-directory>`
  - Config file: `corpusDir: <name-of-directory>`
- Let's look at an example

Let’s fuzz test the Staker contract

- staker/Staker.sol
- **Goal:** Are we actually testing our properties? Does the coverage report provide any insight?

Conclusion

Key takeaways

- **Learned how to write function-level properties!**
  - Pre-condition
  - Action
  - Post-condition
- **Don’t forget about rounding errors when testing arithmetic operations**
- **Test optimization can be done with pre-conditions and arithmetic manipulation**
- **Debugging**
  - Events
  - Coverage reports
- **Internal vs external testing methodologies**
  - External testing is used more in practice

Your homework assignment

Your homework assignment

- **Continue writing properties for ABDK Math!**
  - Associative property of multiplication
    - (x * y) * z = x * (y * z)
  - Distributive property of multiplication
    - x * (y + z) = (x * y) + (x * z)
  - Multiplication of inverses (use the inv() function)
    - inv(x * y) = inv(x) * inv(y)
  - Square roots
    - sqrt(x) * sqrt(x) = x
  - Logarithms
    - log2(x * y) = log2(x) * log2(y)
  - gavg(), pow(), ln(), exp()
- **We will open source these properties (crytic/properties)**

Next streaming session

Next streaming session

- **Part 3 will be on Nov 30, 2022 at 12PM EST**
- **We will be explore the various properties of Uniswap V2!**