Apr 28, 2023 16 min read

# Invariant testing in foundry

Updated: May 12, 2023

## Introduction

In this article, we will discuss invariants and how to perform an invariant test on solidity smart contracts using foundry test suites.

Invariant testing is another test methodology like unit test and fuzzing to verify the correctness of code. If you are unfamiliar with unit tests, please see our article on unit tests using foundry.

To follow up with the practical aspect of this article, you are expected to be familiar with solidity and have foundry installed on your computer. Otherwise, see how to do that here.

### Authorship

This article was co-written by Jesse Raymond (LinkedIn, Twitter) as part of the RareSkills Research and Technical Writing Program.

### Accompanying Repo

If you just want to copy and paste some code, clone the repo we've provided here. You can also use the repo to follow along with this tutorial. https://github.com/RareSkills/invariant-testing-foundry-tutorial

## Why you should test for invariants

Invariant testing allows us to test aspects of a smart contract that unit tests will likely miss. Unit tests only cover properties specified in the test and nothing else. But with invariant testing, smart contracts are tried and tested under multiple random states to find flaws in the code.

By testing these invariants, developers can catch potential issues that unit tests or manual code reviews may not detect.

## What are invariants

Invariants are conditions that must always be true under a certain set of well-defined assumptions. For example, in an ERC20 contract, an invariant would be that the sum of all balances in the contract should equal the total supply. If a function call or transaction violates this invariant, something has gone wrong with the code, and the system is no longer functioning properly.

Whereas unit tests verify specific behavior, invariants say something about the system as a whole. Here are some examples:

- The total supply of an ERC20 token does not change if mint or burn are not called
- The total rewards from a smart contract cannot exceed a certain percentage over a fixed period
- Users cannot withdraw more than they deposit + some capped reward

# Getting started

An invariant test in foundry is a **stateful fuzz test**, where a contract's functions are called randomly with random inputs by the fuzzer, all to try to break any specified invariant. A stateful fuzz test means that the state of the test at one call is saved for the next call.

Let us initialize a new foundry project to perform an invariant test on a smart contract.

Run the following command:

```
forge init invariant-exercise
cd invariant-exercise
```

Now we have our foundry project ready.

# Foundry configs

We can set optional configuration values for our invariant test inside the foundry.toml file. Foundry uses default values if no config values are set. We will set only the important ones as we proceed in this article. To see all available invariant configs visit here.

- runs: The number of runs that must execute for each invariant test group (default value is 256).
- depth: The number of calls executed to attempt to break invariants in one run (default value is 15).
- fail_on_revert: Fails the invariant fuzzing if a revert occurs (default value is false.)

# A simple example

Now rename the Counter.sol that comes with foundry to Deposit.sol and paste this code.

```
contract Deposit {
    address public seller = msg.sender;
    mapping(address => uint256) public balance;

    function deposit() external payable {
        balance[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 amount = balance[msg.sender];
        balance[msg.sender] = 0;
```

```
            (bool s, ) = msg.sender.call{value: amount}("");
            require(s, "failed to send");
        }
    }
```

**This is a simple contract that allows anyone to deposit ether and withdraw them.**

**Deposited ether should always be withdraw-able by the depositor at all times since there are no restrictions.**

**Our invariant should be that any amount deposited should be withdraw-able by the same person and the same amount.**
**We will implement an invariant test to confirm that:**

- **The depositor can withdraw ether deposited.**
- **The same amount deposited would be the same amount withdrawn by the depositor.**

**Let us verify our code is correct by writing an invariant test for both cases.**
**Head over to the test folder in our foundry project, rename the Counter.t.sol to Deposit.t.sol, and paste the code below.**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "../src/Deposit.sol";

contract InvariantDeposit is Test {
    Deposit deposit;

    function setUp() external {
        deposit = new Deposit();
        vm.deal(address(deposit), 100 ether);
    }

     function invariant_alwaysWithdrawable() external payable {
        deposit.deposit{value: 1 ether}();
        uint256 balanceBefore = deposit.balance(address(this));
        assertEq(balanceBefore, 1 ether);
        deposit.withdraw();
        uint256 balanceAfter = deposit.balance(address(this));
        assertGt(balanceBefore, balanceAfter);
    }
```

```
        receive() external payable {}
    }
```

# Explaining the test

**What we are about to do is “Open Testing”. Open testing is where the default configuration for target contracts is set to all contracts deployed inside the test function. You can read further here if desired.**

**Invariants: The depositor can withdraw ether deposited, and the same amount deposited would be the same amount withdrawn by the depositor.**

**The code that verifies that this is correct is the “invariant_alwaysWithdrawable” test function, which is:**

```
    function invariant_alwaysWithdrawable() external payable {
        deposit.deposit{value: 1 ether}();
        uint256 balanceBefore = deposit.balance(address(this));
        assertEq(balanceBefore, 1 ether);
        deposit.withdraw();
        uint256 balanceAfter = deposit.balance(address(this));
        assertGt(balanceBefore, balanceAfter);
    }
```

**Notice that the test function starts with an invariant keyword. This is important because foundry uses this to recognize that this is an invariant test.**

**We start depositing one ether from the test contract. Since the Deposit contract keeps track of the amount deposited through the “balance” mapping, we then use it to take note of our balance immediately after depositing {this should equal one ether since it is what we deposited).**

**Next, we call the withdraw function to take back the ether and also take note of our balance again (it should be zero at this point).**

**This balance note-taking is done with the “balanceBefore” and “balanceAfter” local variables.**

**We expect the amount we deposited to be one ether, so we confirm that with assertEq(balanceBefore, 1 ether);.**

**To confirm that the invariant holds, we expect the “balanceBefore” to be greater than “balanceAfter” since this was our balance when we deposited.**

**To verify this, we use the foundry assertion  assertGt(balanceBefore, balanceAfter);**
**If we run the test with forge test --mt invariant_alwaysWithdrawable , we get the following output:**

```
Running 1 test for test/Deposit.t.sol:InvariantDeposit
[PASS] invariant_alwaysWithdrawable() (runs: 256, calls: 3840, reverts:
1917)
Test result: ok. 1 passed; 0 failed; finished in 347.19ms
```

## Test parameters

**The "runs" parameter refers to the number of times a particular test function is executed. Each time the test function is run, it passes different inputs or conditions to test different scenarios and ensure the contract functions correctly under different conditions.**

**"Calls" refer to the number of times functions in the smart contract are called during a single test run.**

**"Reverts" refers to the number of times a call to any function within the smart contract resulted in a transaction being reverted due to an error or exception.**

# Expecting a revert

**We can see that the test is successful, and the test made calls to the functions of our contract "3840" times in other to break our invariants, as shown in the number of calls.**

**It also reverted "1917" times. This can be where the invariant test or fuzzer tries to call any function in the smart contract without meeting the function's requirements. We will modify our foundry.toml file and add the following invariant test config to confirm this.**

```
[invariant]
fail_on_revert = true
```

**This will make the test fail if there's a revert when trying to break our invariant.**
**Now, we rerun the test with forge test --mt invariant_alwaysWithdrawable, and we get the following:**

```
Test result: FAILED. 0 passed; 1 failed; finished in 8.53ms

Failing tests:
Encountered 1 failing test in test/Deposit.t.sol:InvariantDeposit
[FAIL. Reason: no balance]
        [Sequence]
                sender=0x0000000000000000000000000000000e3d670d7 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=withdraw(), args=[]

 invariant_alwaysWithdrawable() (runs: 1, calls: 1, reverts: 1)

Encountered a total of 1 failing tests, 0 tests succeeded
```

**We can see that the invariant test randomly calls the "withdraw" function from the start, even if we did not specify that (Note that this is entirely random, and you might get a different result on different trials). This is because all the functions of our contract are available for the fuzzer through the open-testing method. We will see how to exclude or include specific contracts/functions when discussing "Invariant targets" later in this article.**

**This random function call attempts to break our invariant by all means. But as specified in the code, the function will revert if the sender has no balance.**
**Because the invariant test behaves this way, we see some revert cases even if our test passes.**

**(Remember to change the fail_on_revert to false, so our test won't stop running)**

# Introducing a vulnerability to the contract for testing

**To test further, let us introduce a vulnerability to the contract that lets anyone change the deposited balance of any address.**

**Add the following code to the Deposit contract:**

```
function changeBalance(address depositor, uint amount) public {
        balance[depositor] = amount;
}
```

**Now we rerun the test with,**

```
forge test --mt invariant_alwaysWithdrawable
```

**and we get the following output:**

```
Test result: FAILED. 0 passed; 1 failed; finished in 74.09ms

Failing tests:
Encountered 1 failing test in test/Deposit.t.sol:InvariantDeposit
[FAIL. Reason: Assertion failed.]
        [Sequence]
                sender=0x0000000000000000000000000000000000000f7a addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x73575ade2424045cf0df8fa1712dde9137c56416 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=changeBalance(address,uint256), args=
[0xba2840574eA60882e96881D1cC3C1d7D90af0e1d, 3]
                sender=0xff1cb1b0420410582bfd4b6b345769b2cc4a51f1 addr=
```

```
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=withdraw(), args=[]
                sender=0x0000000000000000000080808080808149a59da1d addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x6383b40e80395f66de7f61df26bc9bafbbf3cb0f addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=withdraw(), args=[]
                sender=0x97c68648bd6e6ed8a62e640937543f7bf47e39ba addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=changeBalance(address,uint256), args=
[0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496, 2193]

 invariant_alwaysWithdrawable() (runs: 31, calls: 456, reverts: 160)

Encountered a total of 1 failing tests, 0 tests succeeded
```

**Pay attention to the last function call in the call sequence. We can see that the "changeBalance" function is called. The parameters passed here are; 1. The address of the foundry test contract and 2. 2193 (totally random number).**

**This will change the balance of the test contract, which we used in depositing one ether before. So instead of having a balance of one ether, we now have "2193" as our balance. Hence this breaks the invariant that "the same amount deposited would be the same amount withdrawn by the depositor".**

**To confirm that the address passed to the "changeBalance" was the address of the test contract, we can instead impersonate a desired address for the test.**

## But changeBalance() is not in the tests, how did it get called?!

**This is what makes invariant testing awesome. Even though we never explicitly called changeBalance(), the invariant tester threw in calls to it randomly while carrying out the explicit sequence of function calls in the test.**

**This enables invariant testing to test aspects that we "didn't think of."**

## Changing the user's balance instead of the contract's balance

**Let us modify the test function to this:**

```
function invariant_alwaysWithdrawable() external payable {
        vm.startPrank(address(0xaa));
        vm.deal(address(0xaa), 10 ether);
```

```
        deposit.deposit{value: 1 ether}();
        uint256 balanceBefore = deposit.balance(address(0xaa));
        vm.stopPrank();
        assertEq(balanceBefore, 1 ether);

        vm.prank(address(0xaa));
        deposit.withdraw();
        uint256 balanceAfter = deposit.balance(address(0xaa));
        vm.stopPrank();
        assertGt(balanceBefore, balanceAfter);
    }
```

**We are still doing the same thing as before, except that instead of the test contract being the msg.sender, it will be the address(0xaa) we just pranked.**

**Now rerun the test with forge test --mt invariant_alwaysWithdrawable, and we get the following:**

```
Test result: FAILED. 0 passed; 1 failed; finished in 85.64ms

Failing tests:
Encountered 1 failing test in test/Deposit.t.sol:InvariantDeposit
[FAIL. Reason: Assertion failed.]
        [Sequence]
                sender=0x00000000000000000000000000000000000000e6 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x000000000000000000000000000000000090c5013b addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x0000000000000000000000000000000000000001 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=changeBalance(address,uint256), args=
[0x00000000000000000000000000000000000000A1, 296312983667185193009]
                sender=0x000000000000000000000000000000000000000c addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=withdraw(), args=[]
                sender=0x0000000000000000000000000000000000000009 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x00000000000000000000000000000000000000fc5 addr=
```

```
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x00000000000000000000000000000000000005fb addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=withdraw(), args=[]
                sender=0x0000000000000000000000000000000000000005 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0xb30de0face1af7a50fbd59f1a0d9f31e9282d40f addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=deposit(), args=[]
                sender=0x0000000000000000000000000000000000000a94 addr=
[src/Deposit.sol:Deposit]0x5615deb798bb3e4dfa0139dfa1b3d433cc23b72f
calldata=changeBalance(address,uint256), args=
[0x00000000000000000000000000000000000000AA, 4594637]

 invariant_alwaysWithdrawable() (runs: 2, calls: 33, reverts: 8)

Encountered a total of 1 failing tests, 0 tests succeeded
```

**The same action was repeated, but this time with address(0xaa) (as we can see in the last call sequence) and not the address of the test contract.**

**Logically, it also breaks the first invariant: “The depositor can withdraw ether deposited". The “changeBalance” function we introduced can be called with any address and zero as the amount to change the balance.**

**This will make that address which has presumably deposited before, now have zero balance and thus cannot withdraw even if their ether is in the contract.**

# Conditional invariants

**While invariants are to hold at all times, some invariants require certain conditions to hold. For example, an invariant such as assertEq(token.totalSupply(), 0); should only hold when there has been no mint. The total supply would not be zero if the token were minted.**

**These invariants are called conditional invariants because the protocol or smart contract must be under some conditions before they must hold. To find out more, you can check it out here.**

# Changing the invariant test config

**If we want to increase the number of runs for each test, we can add the configs in the foundry.toml file, as stated earlier in this article.**

**Add the following in the foundry.toml file below the [invariant] section.**

```
[invariant] #invariant section
fail_on_revert = false
runs = 1215
depth = 23
```

**Now rerun the test with forge test --mt invariant_alwaysWithdrawable (make sure to have removed or commented out the changeBalance function).**

```
Running 1 test for test/Deposit.t.sol:InvariantDeposit
[PASS] invariant_alwaysWithdrawable() (runs: 1215, calls: 27945,
reverts: 13965)
Test result: ok. 1 passed; 0 failed; finished in 4.39s
```

**The test still passes, but this time the number of runs, calls, and revert is significantly higher than usual because we have modified it in our config. You can choose to use any number from zero to uint32.max.**

**If we set the runs parameter to a number greater than uint32, foundry would throw an error when we try to run the test.**

**For example, let's set it to 23000000000000 and try to run the test.**
**We get this:**

```
Error:
failed to extract foundry config:
foundry config error: invalid value signed int `23000000000000`,
expected u32 for setting `invariant.depth`
```

**A larger number means more test scenarios, but larger numbers make the test slower.**

# Near real-life examples

**We have covered at least the basics of invariant testing with foundry with our contract, but let us go further and perform an invariant test on a popular contract.**

**We'll be testing the SideEntranceLenderPool contract, which is the contract of the fourth level in the popularly known Damn Vulnerable DeFi CTF.**

**This is the contract below:**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;
import "openzeppelin-contracts/contracts/utils/Address.sol";

interface IFlashLoanEtherReceiver {
    function execute() external payable;
}

/**
 * @title SideEntranceLenderPool
 * @author Damn Vulnerable DeFi (https://damnvulnerabledefi.xyz)
 */contract SideEntranceLenderPool {
    using Address for address payable;

    mapping(address => uint256) private balances;
    uint256 public initialPoolBalance;

    constructor() payable {
        initialPoolBalance = address(this).balance;
    }

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw() external {
        uint256 amountToWithdraw = balances[msg.sender];
        balances[msg.sender] = 0;
        payable(msg.sender).sendValue(amountToWithdraw);
    }

    function flashLoan(uint256 amount) external {
        uint256 balanceBefore = address(this).balance;
        require(balanceBefore >= amount, "Not enough ETH in balance");

        IFlashLoanEtherReceiver(msg.sender).execute{value: amount}();
```

```
            require(
                address(this).balance >= balanceBefore,
                "Flash loan hasn't been paid back"
            );
        }
    }
```

**The contract has been modified a little (also note the openzeppelin import) to fit in our foundry project and what we want. We have also installed the necessary dependencies (the OpenZeppelin Address library import).**

**This contract is vulnerable in the "flashLoan" function that lets someone exploit it and drain its ether balance. An attacker could call the "flashLoan" function to take a loan and deposit the same loan back into the contract with the "deposit" function as the attacker's balance, later they can withdraw the balance and get away with it even if it was originally a loan and not their ether.**

**So what's going to be our invariant here?**

**First, it is important to note that the contract has a payable constructor, and the ether used for loans is deposited during deployment. There is also no way for that initially deposited ether to be withdrawn. Ether can only be added to the contract using the "deposit" function and withdrawn with the "withdraw" function (only if the function caller has deposited prior).**

**So if we have this in mind, we can say that the invariant would be**

```
assert(address(SideEntranceLenderPool).balance >=
SideEntranceLenderPool.initialPoolBalance());
```

**("initialPoolBalance" is a public state variable used to store how much ether was deposited during deployment).**

**We assert that the "SideEntranceLenderPool" ether balance is always greater than or equal to the ether deposited during deployment.**

**If everything works fine, this invariant should hold. But as said earlier, a vulnerability allows someone to deposit a loan taken from the contract and withdraw it later.**
**In the next section, we will introduce a new concept in foundry invariant testing called a *Handler* to achieve better results.**

# Handler-based testing

**A handler contract is used to test more complex protocols or contracts. It works as a wrapper contract that will be used to interact or make calls to our desired contract.**

**It is particularly necessary when the environment needs to be configured in a certain way (i.e. a constructor is called with certain parameters).**

**How it works is that, in the "setUp" function in the test file, we deploy the handler contract that will make calls to the pool contract and set only this handler contract as the target contract in the test using the targetContract(address target) test helper function.**

**Because of this, only the functions of the handler contract would be called randomly by the fuzzer.**

**Another benefit is that if a function in the main contract (SideEntranceLenderPool contract in this case) requires a certain condition before it can be called, we can easily define it in the handler contract before the function call.**

**The handler contract can also inherit the forge-std Test and use foundry cheatsheets like vm.deal, vm.prank, etc. We will demonstrate this as we go.**

**Let's create a /handler folder inside the test folder and a handler.sol file inside it.**
**This will be the code for our handler contract.**

```
import {SideEntranceLenderPool} from
"../../src/SideEntranceLenderPool.sol";

import "forge-std/Test.sol";

contract Handler is Test {
    // the pool contract
    SideEntranceLenderPool pool;

    // used to check if the handler can withdraw ether after the
exploit
    bool canWithdraw;

    constructor(SideEntranceLenderPool _pool) {
        pool = _pool;

        vm.deal(address(this), 10 ether);
    }

    // this function will be called by the pool during the flashloan
    function execute() external payable {
        pool.deposit{value: msg.value}();
        canWithdraw = true;
```

```
        }

        // used for withdrawing ether balance in the pool
        function withdraw() external {
            if (canWithdraw) pool.withdraw();
        }

        // call the flashloan function of the pool, with a fuzzed amount
        function flashLoan(uint amount) external {
            pool.flashLoan(amount);
        }

        receive() external payable {}
    }
```

**We have defined functions in the handler contract that call the "SideEntranceLenderPool" contract functions. This is so we can test more edge cases and practically exploit the vulnerability.**

**The handler contract inherits the forge-std Test as stated earlier, and the vm.deal method is used inside the constructor of the handler contract to give the contract some ether.**

## Invariant Targets and test helpers

**Foundry comes with test helper functions in the forge-std library that allows us to specify our target contracts, target artifacts, target selectors, and target artifacts selectors.**
**Some helper functions are**

- **targetContract(address newTargetedContract_)**
- **targetSelector(FuzzSelector memory newTargetedSelector_)**
- **excludeContract(address newExcludedContract_).**

**To see all available test helper functions, see here and here.**

**We will create a SideEntranceLenderPool.t.sol test file inside the test folder. This is where we'll define our invariant test for the SideEntranceLenderPool contract and specify the handler contract as our invariant target.**

**Paste the following code inside the test file:**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "forge-std/Vm.sol";
import "forge-std/console2.sol";
```

```
import "../src/SideEntranceLenderPool.sol";
import "./handlers/Handler.sol";

contract InvariantSideEntranceLenderPool is Test {
    SideEntranceLenderPool pool;
    Handler handler;

    function setUp() external {
        // deploy the pool contract with 25 ether
        pool = new SideEntranceLenderPool{value: 25 ether}();
        // deploy the handler contract
        handler = new Handler(pool);
        // set the handler contract as the target for our test
        targetContract(address(handler));
    }

    // invariant test function
    function invariant_poolBalanceAlwaysGtThanInitialBalance() external
{
        // assert that the pool balance will never go below the initial
balance (the 10 ether deposited during deployment)
        assert(address(pool).balance >= pool.initialPoolBalance());
    }
}
```

**After pasting the code, let's run the test with**

```
forge test --mt invariant_poolBalanceAlwaysGtThanInitialBalance
```

**We get this output:**

```
Test result: FAILED. 0 passed; 1 failed; finished in 19.08ms

Failing tests:
Encountered 1 failing test in
test/SideEntranceLenderPool.t.sol:InvariantSideEntranceLenderPool
[FAIL. Reason: Assertion violated]
        [Sequence]
                sender=0x0000000000000000000000000000000000000531 addr=
[test/handlers/Handler.sol:Handler]0x2e234dae75c793f67a35089c9d99245e1c
```

```
58470b calldata=flashLoan(uint256), args=[3041954473]
                sender=0x0000000000000000000000000000000000000423 addr=
[test/handlers/Handler.sol:Handler]0x2e234dae75c793f67a35089c9d99245e1c
58470b calldata=withdraw(), args=[]

 invariant_poolBalanceAlwaysGtThanInitialBalance() (runs: 1, calls: 8,
reverts: 0)
```

**The test was able to break the invariant and find the exploit.**

**The "flashLoan" function was called first, and then the "withdraw" function.**
**To see the complete stack trace and call sequence, we can rerun the test with**

```
forge test --mt invariant_poolBalanceAlwaysGtThanInitialBalance -vvvv
```

```
[45514] Handler::flashLoan(3041954473)
    ├─ [40246] SideEntranceLenderPool::flashLoan(3041954473)
    │   ├─ [32885] Handler::execute{value: 3041954473}()
    │   │   ├─ [22437] SideEntranceLenderPool::deposit{value:
3041954473}()
    │   │   │   └─ ← ()
    │   │   └─ ← ()
    │   └─ ← ()
    └─ ← ()

  [14076] Handler::withdraw()
    ├─ [9828] SideEntranceLenderPool::withdraw()
    │   ├─ [55] Handler::receive{value: 3041954473}()
    │   │   └─ ← ()
    │   └─ ← ()
    └─ ← ()

    [7724]
InvariantSideEntranceLenderPool::invariant_poolBalanceAlwaysGtThanIniti
alBalance()
    ├─ [2261] SideEntranceLenderPool::initialPoolBalance() [staticcall]
```

```
        │   └─ ← 25000000000000000000 #initial balance was 25 ether
        └─ ← "Assertion violated"
```

**Now we can visualize the whole call sequence and see how the invariant was broken.**

# An example with a mathematical statement

**This example will be a stateless fuzz, i.e. the behavior doesn't depend on previous calls. The intent here is to demonstrate limitations of fuzzing and how to work around it. We could add some storage variables to turn this into a stateful fuzz, but that would be a distraction for now.**

**Here is our example contract:**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
contract Quadratic {
    bool public ok = true;

    function notOkay(int x) external {
        if ((x - 11111) * (x - 11113) < 0) {
            ok = false;
        }
    }
}
```

**This is a straightforward one, and we are just going to test that "ok" boolean variable is true at all times, that is; assertTrue(quadratic.ok());**
**It only becomes false if the "notOkay" function is called with a number that fulfills this statement (x - 11111) * (x - 11113) < 0.**

**This might look easy, but let's see if the fuzzer can find a number and break the invariant.**

**We would also use the handler method here, so create a Handler_2.sol file inside the /test/handler folder and paste this code.**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "../../src/Quadratic.sol";
import "forge-std/Test.sol";

contract Handler_2 is Test {
    Quadratic quadratic;
```

```
    constructor(Quadratic _quadratic) {
        quadratic = _quadratic;
    }

    function notOkay(int x) external {
        quadratic.notOkay(x);
    }
}
```

**Now create a Quadratic.t.sol file inside the test folder and paste this code inside:**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "./handlers/Handler_2.sol";
import "../src/Quadratic.sol";

contract InvariantQuadratic is Test {
    Quadratic quadratic;
    Handler_2 handler;

    function setUp() external {
        quadratic = new Quadratic();
        handler = new Handler_2(quadratic);

        targetContract(address(handler));
    }

    function invariant_NotOkay() external {
        assertTrue(quadratic.ok());
    }
}
```

**We have defined our invariant in the invariant_NotOkay function.**
**Run the test with**

```
forge test --mt invariant_NotOkay
```

**and we get:**

```
Running 1 test for test/Quadratic.t.sol:InvariantQuadratic
[PASS] invariant_NotOkay() (runs: 256, calls: 3840, reverts: 760)
Test result: ok. 1 passed; 0 failed; finished in 576.70ms
```

**The test passed and the fuzzer couldn't break the invariant. But a number that will break this invariant exists, and we will show what it is later, but for now, let us increase the number of runs for the test to see if it finds it.**

**Set the number of runs to 20,000.**

```
[invariant]
runs = 20000
```

**We reran the test and got the following:**

```
Running 1 test for test/Quadratic.t.sol:InvariantQuadratic
[PASS] invariant_NotOkay() (runs: 20000, calls: 300000, reverts: 74275)
Test result: ok. 1 passed; 0 failed; finished in 92.41s
```

**Even with a high run, the test couldn't break the invariant, passing a number that makes "ok" false.**

**To confirm that a number exists, the desmos graph shows this number when the equation is inputted, as shown in the blue circled part in the image below.**

**The image shows that the number we need is "11112".**

**Let's try bounding the range of numbers that the fuzzer will use on the handler contract with x = bound(x, 11_000, 100_000);. Add this line of code in the "notOkay" function of the handler contract (the second handler contract).**
**It should now look like this:**

```
function notOkay(int x) external {
    x = bound(x, 10_000, 100_000);
    quadratic.notOkay(x);
}
```

**The bound helper function comes with the forge-std Test library; we can limit the range of fuzzed inputs.**

**Rerun the test with**

```
forge test --mt invariant_NotOkay -vvv
```

**and we get the following:**

```
Test result: FAILED. 0 passed; 1 failed; finished in 20.49s

Failing tests:
Encountered 1 failing test in test/Quadratic.t.sol:InvariantQuadratic
[FAIL. Reason: Assertion failed.]
        [Sequence]
                sender=0x0000000000000000000000000000000001373a addr=
[test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
5e1c58470b calldata=notOkay(int256), args=[-5675015641267]
                sender=0x0000000000000000000000000000000000002df6 addr=
[test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
5e1c58470b calldata=notOkay(int256), args=[-3]
                sender=0x0000000000000000000000000000000000009208 addr=
[test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
5e1c58470b calldata=notOkay(int256), args=
[191219569823024188795377493431890629903 6]
                sender=0x000000000000000000000000000000000000172fd addr=
[test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
5e1c58470b calldata=failed():(bool), args=[]
                sender=0x41b9a90e4836f4df4fe8ed9933c618c49163d8c3 addr=
[test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
5e1c58470b calldata=failed():(bool), args=[]
```

```
                sender=0x0000000000000000000000000000000000005001 addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=notOkay(int256), args=
    [57896044618658097711785492504343953926634992332820282019728792003956564819794]
                sender=0x000000000000000000000000000000000000e860 addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=failed():(bool), args=[]
                sender=0x2039383034370000000000000000000000000000 addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=notOkay(int256), args=
    [513761924256431362626206017641167949844669773357 0]
                sender=0x000000000000000000000000000000000008ead addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=failed():(bool), args=[]
                sender=0x2d4326d8f5a6b7c3ef871eb0063dc7771fd571d8 addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=failed():(bool), args=[]
                sender=0xc7ebe193ccfed949da23e957c37020d88a068c34 addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=notOkay(int256), args=
    [57896044618658097711785492504343953926634992332813620401282714769779013280756]
                sender=0xd72485927db413065ce2730222fc574be7f38a83 addr=
    [test/handlers/Handler_2.sol:Handler_2]0x2e234dae75c793f67a35089c9d9924
    5e1c58470b calldata=notOkay(int256), args=
    [-57896044618658097711785492504343953926634992332820282019728792003956564809711]

     invariant_NotOkay() (runs: 3470, calls: 52047, reverts: 0)

    Encountered a total of 1 failing tests, 0 tests succeeded
```

**The invariant did break the assertion this time because of the limited range of fuzzed inputs, but looking at the call sequence, we can't see where the "notOkay" function was called with "11112".**

**We used the verbosity flag -vvv to see what was happening.**

**There are also log traces in the test result, which are:**

```
 invariant_NotOkay() (runs: 3470, calls: 52047, reverts: 0)
Logs:
  Bound result 23762
  Bound result 89998
  Bound result 44363
  Bound result 88972
  Bound result 11664
  Bound result 33484
  Bound result 11112

Traces:
  [14840] Handler_2::notOkay(-5675015641267)
    ├─ [0] VM::toString(23762) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000002000000
0000000000000000000000000000000000000000000000000000053233373632000
0000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 23762) [staticcall]
    │   └─ ← ()
    ├─ [607] Quadratic::notOkay(23762)
    │   └─ ← ()
    └─ ← ()

  [14840] Handler_2::notOkay(-3)
    ├─ [0] VM::toString(89998) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000002000000
0000000000000000000000000000000000000000000000000000053839393800
0000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 89998) [staticcall]
    │   └─ ← ()
    ├─ [607] Quadratic::notOkay(89998)
    │   └─ ← ()
    └─ ← ()

  [14772] Handler_2::notOkay(191219569823024188795377493431890629903 6)
    ├─ [0] VM::toString(44363) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000002000000
0000000000000000000000000000000000000000000000000000053434333633000
```

```
0000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 44363) [staticcall]
    │   └─ ← ()
    ├─ [607] Quadratic::notOkay(44363)
    │   └─ ← ()
    └─ ← ()

  [14772]
Handler_2::notOkay(57896044618658097711785492504343953926634992332820282019728792003956564819794)
    ├─ [0] VM::toString(88972) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000053838393732000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 88972) [staticcall]
    │   └─ ← ()
    ├─ [607] Quadratic::notOkay(88972)
    │   └─ ← ()
    └─ ← ()

  [14772]
Handler_2::notOkay(513761924256431362626206017641167949844669773357 0)
    ├─ [0] VM::toString(11664) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000053131363634000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 11664) [staticcall]
    │   └─ ← ()
    ├─ [607] Quadratic::notOkay(11664)
    │   └─ ← ()
    └─ ← ()

  [14772]
Handler_2::notOkay(57896044618658097711785492504343953926634992332813620401282714769779013280756)
    ├─ [0] VM::toString(33484) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000000002000000
```

```
0000000000000000000000000000000000000000000000005333334383400
0000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 33484) [staticcall]
    │   └─ ← ()
    ├─ [607] Quadratic::notOkay(33484)
    │   └─ ← ()
    └─ ← ()


  [15887]
Handler_2::notOkay(-57896044618658097711785492504343953926634992332820282
8201972879200395656480971l)
    ├─ [0] VM::toString(11112) [staticcall]
    │   └─ ←
0x0000000000000000000000000000000000000000000000000000000002000000
0000000000000000000000000000000000000000000000005313131313200
0000000000000000000000000000000000000000000000000
    ├─ [0] console::log(Bound result, 11112) [staticcall]
    │   └─ ← ()
    ├─ [4500] Quadratic::notOkay(11112)
    │   └─ ← ()
    └─ ← ()
```

**The log traces show that the fuzzer called the “notOkay” function with numbers that are not even close to the desired number. Still, the bound function kept changing these inputs until the right number was gotten, as shown in the last bound result and call sequence.**

**Using the bound function in cases where a particular range of numbers should be tested can be handy and helps to achieve better results.**

# Conclusion

**In this article, we learned what invariants are, why they are important, and how to perform invariant testing in a foundry.**

**We also discussed conditional invariants, handler-based setup, and how and when to bound the range of fuzzed’s input values.**

# Learn More

**Our advanced solidity training teaches modern smart contract testing beyond unit tests. Check it out to learn more.**