This article will teach you everything you need to know to understand a key testing strategy, crucial when it comes to ensuring the reliability of your systems:

Smart contracting Fuzzing (fuzz testing) and invariant testing.

Throughout this guide and its examples, we'll explore how we can test our invariants

and smart contract functions using fuzz testing and Foundry while addressing and uncovering complex and often subtle aspects of smart contract behaviours.

Want to level up your smart contract security skills

? Take our 20+ hours smart contract auditing course on Cyfrin Updraft, completely free!

Here's a video with Troy, Trail of Bits security engineer, explaining what is fuzzing and how it works.

The first thing to understand why fuzzing is important, is understanding what an invariant is.

Let's find out.

What is an invariant?

Smart contract testing includes various methods to enhance smart contracts' on-chain security and efficiency. We test to ensure that a smart contract always behaves as expected across different scenarios to avoid unexpected events and exploits.

Invariant testing involves defining a set of conditions - invariants

- that must always hold, regardless of the contract's state or input.

Invariant testing always requires:

- Invariant definition

: Identifying crucial conditions that must always remain true for proper contract functionality.

- State analysis:

This process rigorously examines the contract under diverse states and input scenarios. The aim is to ensure that the defined invariants remain valid and un-breached throughout these conditions.

Note

: Foundry often categorizes an invariant test as a stateful fuzz test.

In DeFi protocols, for example, a good invariant might be:

- The protocol must always be over-collateralized
- A user should never be able to withdraw more money than they deposited
- There can only be 1 winner of the fair lottery

Once we have found our invariants, it's time to test them and test the thesis that "they should always hold", hence, they should never change status.

One way we can test them is through simple unit tests, but there's a problem, that fuzz testing solves, you'll see in the next section.

Why isn't unit testing invariants using Foundry a good idea?

Let's take into consideration a simple function in a smart contract called SimpleStorage

.

The function setAddition

multiplies any input number by 2

.

Our invariant or “property of the system that should always hold”,

in this case, will be:

- storedData

should never revert

Simply put, we could say our invariant is:

You can find a list of invariants (properties) of popular smart contract standards in this GitHub repo.

Now, using Foundry, let’s write a traditional unit test for this contract, and check that our invariant holds:

In this testing scenario, we initialize a test variable named testValue

to pass to our setAddition function and assign it a value of 100

.

After executing the function, we use assert to make sure the function has returned successfully:

Nice! In this case, our unit invariant test using Foundry tells us that a value of 100

won't break the invariant.

What is not telling us, though, is that any value from 128

and above will indeed break our invariant, overflowing our uint8 storedData variable:

If we rerun the test, this will make the transaction revert and throw the overflow error:

As you can see, unit testing invariants aren't effective. In non-obvious situations, configuring parameters individually to identify every case under which a value testValue

triggers an "arithmetic underflow or overflow" error is time-consuming, if not impossible, or inconvenient.

That’s where fuzz testing or fuzzing, comes in. Let's understand what fuzz testing is.

What is fuzzing (fuzz testing)?

Fuzz testing lifts the need to manually test every possible value a variable, such as testValue

, might assume.

So, what is fuzz testing?

Having that said, there are two types of fuzz testing: stateless and stateful fuzzing.

How does stateless Fuzzing work?

Stateless fuzzing

would be similar to doing something to a balloonA

in one random attempt to break it, then blowing up a new balloonB

and attempting to break it differently.

Here, you’d never try to break a balloon you already tried to break in the past. This seems a little silly as if our balloon invariant is that “the balloon can’t be popped” we’d want to make multiple attempts on the same balloon.

We can see an example of a stateless fuzz test using Foundry in the testFuzzRevertOnOverflow example below:

In this example, we’re not declaring the testValue

variable anymore. Instead, we pass it as an argument to the function, that Foundry will generate and pass at runtime.

As soon as we run our test, it will fail, indicating the input used to break the test:

As you can see, together with the number used to break our function, there's another parameter: "runs" - let's see what it means.

Runs and counterexamplesThe runs count in the example above, indicates the number of randomly generated inputs. In this example, our fuzzer had to generate 3 random different inputs for testValue to find a value that breaks our invariant.

The value that our fuzz test found as an example that breaks our invariant is known as Counterexample

an input that breaks our property or invariant.

How did the fuzz test find the counterexample with 3 runs?

- It ran the unit test with some number, and that number passed the test.
- it chose another number
- It chose 128 and it failed!

Because our fuzz test tried 3 different numbers, we say it did 3 runs. Sometimes, a fuzz test will try thousands of potential counterexamples, and none of them will work because there isn't a bug! When that happens, a developer might end up waiting forever.

We usually give our tool a maximum number of runs to mitigate this. In Foundry, we can find the maximum number of runs in the foundry.toml

As we said before, this test is categorized as stateless because the storedData

variable (our balloon) is reset to its default value of 0 after each iteration.

How does Stateful Fuzzing work?

On the other hand, when the variable's state is retained across multiple runs, such as the counter

value in this scenario, the test transforms into a stateful fuzz test

.

So in the balloon example, we will make different attempts to break it but use the same balloon on each iteration.

In another smart contract scenario, let's establish a new invariant named alwaysEvenNumber

. The core condition of this invariant is that the number must always be even, never odd.

To illustrate the concept, we add a new variable called hiddenValue

. This variable's value is added in the function and equals to the inputNumber value.

If hiddenValue

equals 8, then alwaysEvenNumber

is set to 3. This action violates the invariant as it's supposed to maintain an even number status:

In a stateless setup, where hiddenValue

resets to 0

for each execution, the specific condition we've set will not activate.

However, similar to our balloon analogy, in a stateful fuzz test, we repeatedly use the same "balloon", where the variable's previous value is retained, allowing us to uncover potentially more systemic issues.

Unlike stateless fuzzing, the test will not require us to set up the random input parameter on the function declaration.

Instead, by importing StdInvariant.sol

we can set a targetContract

, Foundry will then take care of automatically triggering random functions of the contract using random parameters and remember its previous state.

Here’s how we would write this test.

AlwaysEven just has one function, so it will just call setEvenNumber with different values and assert if the value returned by the function will remain even.

As we can notice, running our stateful fuzz test using foundry, we were able to randomly values to the function, until 8 was used as an argument changing the AlwaysEven value to 3, ultimately breaking our invariant rule. Again, this is a very simple use case, but in real-world applications finding the value that will break an invariant won’t be that intuitive, making stateful fuzz tests the fastest, sometimes the only, solution.

A note on different test types of fuzz and invariant testing

When it comes different types of tests, terminology can be confusing.

For instance, Foundry often categorizes an invariant test as a stateful fuzz test

, even though we have demonstrated testing an invariant using a stateless fuzz test

in this article.

Additionally, while the term invariant

can be used in the context of a stateful test, it's not mandatory and even confusing sometimes, because as you saw we created a unit test also using invariants.

To clarify these distinctions, here's a detailed graph by Nisedo outlining the various test types:

Best Fuzz testing (fuzzing) tools for smart contract auditors

To conclude, here's a list of the best fuzzers, fuzz testing tools, forsmart contract auditors.

Echidna: Focused on Fuzz Testing

Echidna, developed by Trail of Bits for Ethereum smart contracts, specializes in fuzz testing and is known for its efficacy in vulnerability detection.

- Functionalities

: Generates diverse inputs to scrutinize contract functions, revealing concealed flaws.

- User Accessibility

: Designed for ease of use, accommodating developers of varying expertise.

- CI Integration Capability

: Facilitates integration into continuous integration workflows for ongoing testing.

Medusa: Emerging Smart Contract Fuzzer

Medusa is a Go-Ethereum-based fuzzer for smart contracts, inspired by Echidna. It enables parallelized fuzz testing through its CLI and Go API, facilitating a user-extended testing approach.

- Functionalities

: Medusa supports parallel fuzzing across multiple workers, built-in assertion and property testing, mutational value generation, and coverage collecting.

- Coverage Techniques

: Implements coverage-guided fuzzing, using increasing call sequences from a corpus to guide the fuzzing campaign.

- Extensibility

: Offers an extensible low-level testing API with events and hooks, although the high-level testing API is still under development.

Foundry: Versatile Testing Suite

Foundry stands out for its comprehensive support of both fuzz and invariant testing and is the framework we used for all of the examples above.

- Fuzz Testing Tools

: Simplifies writing and executing fuzz tests seamlessly, integrating them with unit testing.

- Invariant Testing Features

: Equipped for defining and validating invariants during tests.

- Community and Documentation Support

: Extensive resources and an active user base make it a preferred choice for developers.

Conclusion

Adopting fuzz and invariant testing transcends standard practice in smart contract development—it's a vital necessity.

In this article you've learned what is fuzz testing (fuzzing) and how it works. If you want to implementfuzz tests using Foundry, checkout our full tutorial.

These methods highly elevate contract security and reliability in the blockchain realm. Fuzz testing, with tools like Echidna, exposes contracts to a vast range of inputs, unearthing hidden vulnerabilities. Invariant testing, facilitated by platforms like Foundry, ensures contracts uphold logical consistency under various conditions.

This is the new floor for security in web3, if you want to become atop-tier, best-paid auditor in the industry you must understand and apply techniques like this one on a daily basis and the best Blockchain Developers out there must also nail these techniques to build more robust, secure and reliable protocols.