---

title: Testing smart contracts description: An overview of techniques and considerations for testing Ethereum smart contracts. lang: en

---

Public blockchains like Ethereum are immutable, making it difficult to change a smart contracts code after deployment. Contract upgrade patterns for performing "virtual upgrades" exist, but these are difficult to implement and require social consensus. Moreover, an upgrade can only fix an error *after* it is discovered—if an attacker discovers the vulnerability first, your smart contract is at risk of an exploit.

For these reasons, testing smart contracts beforedeploying to Mainnet is a minimum requirement forsecurity. There are many techniques for testing contracts and evaluating code correctness; what you choose depends on your needs. Nevertheless, a test suite made up of different tools and approaches is ideal for catching both minor and major security flaws in contract code.

Prerequisites {#prerequisites}

This page explains how to test smart contracts before deploying on the Ethereum network. It assumes you're familiar with smart contracts.

What is smart contract testing? {#what-is-smart-contract-testing}

Smart contract testing is the process of verifying that the code of a smart contract works as expected. Testing is useful for checking if a particular smart contract satisfies requirements for reliability, usability, and security.

Although approaches vary, most testing methods require executing a smart contract with a small sample of the data it is expected to handle. If the contract produces correct results for sample data, it is assumed to be functioning properly. Most testing tools provide resources for writing and executing test cases to check if a contracts execution matches the expected results.

Why is it important to test smart contracts? {#importance-of-testing-smart-contracts}

As smart contracts often manage high-value financial assets, minor programming errors can and often lead tomassive losses for users. Rigorous testing can, however, help you discover defects and issues in a smart contract's code early and fix them before launching on Mainnet.

While it is possible to upgrade a contract if a bug is discovered, upgrades are complex and canresult in errors if handled improperly. Upgrading a contract further negates the principle of immutability and burdens users with additional trust assumptions. Conversely, a comprehensive plan for testing your contract mitigates smart contract security risks and reduces the need to perform complex logic upgrades after deploying.

Methods for testing smart contracts {#methods-for-testing-smart-contracts}

Methods for testing Ethereum smart contracts fall under two broad categories:**automated testing** and **manual testing**. Automated testing and manual testing offer unique benefits and tradeoffs, but you can combine both to create a robust plan for analyzing your contracts.

Automated testing {#automated-testing}

Automated testing uses tools that automatically check a smart contracts code for errors in execution. The benefit of automated testing comes from using scripts to guide the evaluation of contract functionalities. Scripted tests can be scheduled to run repeatedly with minimal human intervention, making automated testing more efficient than manual approaches to testing.

Automated testing is particularly useful when tests are repetitive and time-consuming; difficult to carry out manually; susceptible to human error; or involve evaluating critical contract functions. But automated testing tools can have drawbacks—they may miss certain bugs and produce many false positives. Hence, pairing automated testing with manual testing for

smart contracts is ideal.

Manual testing {#manual-testing}

Manual testing is human-aided and involves executing each test case in your test suite one after the other when analyzing a smart contracts correctness. This is unlike automated testing where you can simultaneously run multiple isolated tests on a contract and get a report showing all failing and passing tests.

Manual testing can be carried out by a single individual following a written test plan that covers different test scenarios. You could also have multiple individuals or groups interact with a smart contract over a specified period as part of manual testing. Testers will compare the actual behavior of the contract against the expected behavior, flagging any difference as a bug.

Effective manual testing requires considerable resources (skill, time, money, and effort), and it is possible—due to human error—to miss certain errors while executing tests. But manual testing can also be beneficial—for example, a human tester (e.g., an auditor) may use intuition to detect edge cases that an automated testing tool would miss.

Automated testing for smart contracts {#automated-testing-for-smart-contracts}

Unit testing {#unit-testing-for-smart-contracts}

Unit testing evaluates contract functions separately and checks that each component works correctly. Good unit tests should be simple, quick to run and provide a clear idea of what went wrong if tests fail.

Unit tests are useful for checking that functions return expected values and that contract storage is updated properly after function execution. Moreover, running unit tests after making changes to a contracts codebase ensures adding new logic does not introduce errors. Below are some guidelines for running effective unit tests:

Guidelines for unit testing smart contracts {#unit-testing-guidelines}

1. Understand your contracts business logic and workflow

Before writing unit tests, it helps to know what functionalities a smart contract offers and how users will access and use those functions. This is particularly useful for running happy path tests that determine if functions in a contract return the correct output for valid user inputs. We'll explain this concept using this (abridged) example of an auction contract

``` constructor( uint biddingTime, address payable beneficiaryAddress ) { beneficiary = beneficiaryAddress; auctionEndTime = block.timestamp + biddingTime; }

function bid() external payable {

```
  if (block.timestamp > auctionEndTime)
        revert AuctionAlreadyEnded();

  if (msg.value <= highestBid)
        revert BidNotHighEnough(highestBid);
```

if (highestBid != 0) { pendingReturns[highestBidder] += highestBid; } highestBidder = msg.sender; highestBid = msg.value; emit HighestBidIncreased(msg.sender, msg.value); }

function withdraw() external returns (bool) { uint amount = pendingReturns[msg.sender]; if (amount > 0) { pendingReturns[msg.sender] = 0;

```
    if (!payable(msg.sender).send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
    return true;
}
```

function auctionEnd() external { if (block.timestamp < auctionEndTime) revert AuctionNotYetEnded(); if (ended) revert
```

AuctionEndAlreadyCalled();

```
    ended = true;
    emit AuctionEnded(highestBidder, highestBid);

    beneficiary.transfer(highestBid);
}

} ```
```

This is a simple auction contract designed to receive bids during the bidding period. If the `highestBid` increases, the previous highest bidder receives their money; once the bidding period is over, the `beneficiary` calls the contract to get their money.

Unit tests for a contract like this would cover different functions a user might call when interacting with the contract. An example would be unit a test that checks if a user can place a bid while the auction is ongoing (i.e., calls to `bid()` succeed) or one that checks if a user can place a higher bid than the current `highestBid`.

Understanding a contracts operational workflow also helps with writing unit tests that check if execution meet requirements. For example, the auction contract specifies that users cannot place bids when the auction has ended (i.e., when `auctionEndTime` is lower than `block.timestamp`). Thus, a developer might run a unit test that checks if calls to the `bid()` function succeed or fail when the auction is over (i.e., when `auctionEndTime` > `block.timestamp`).

2. Evaluate all assumptions related to contract execution

It is important to document any assumptions about a contract's execution and write unit tests to verify the validity of those assumptions. Apart from offering protection against unexpected execution, testing assertions forces you to think about operations that could break a smart contracts security model. A useful tip is to go beyond "happy user tests" and write negative tests that check if a function fails for the wrong inputs.

Many unit testing frameworks allow you to create assertions—simple statements that state what a contract can and cannot do—and run tests to see if those assertions hold under execution. A developer working on the auction contract described earlier could make the following assertions about its behavior before running negative tests:

- Users cannot place bids when the auction is over or hasn't begun.
- The auction contract reverts if a bid is below the acceptable threshold.
- Users who fail to win the bid are credited with their funds

Note: Another way of testing assumptions is to write tests that trigger function modifiers in a contract, especially `require`, `assert`, and `if…else` statements.

3. Measure code coverage

Code coverage is a testing metric that tracks the number of branches, lines, and statements in your code executed during tests. Tests should have good code coverage, otherwise you may get "false negatives" which happen a contract passes all tests, but vulnerabilities still exist in the code. Recording high code coverage, however, gives the assurance all statements/functions in a smart contract were sufficiently tested for correctness.

4. Use well-developed testing frameworks

The quality of the tools used in running unit tests for your smart contracts is crucial. An ideal testing framework is one that's regularly maintained; provides useful features (e.g., logging and reporting capabilities); and must have been extensively used and vetted by other developers.

Unit testing frameworks for Solidity smart contracts come in different languages (mostly JavaScript, Python, and Rust). See some of the guides below for information on how to start running unit tests with different testing frameworks:

- **Running unit tests with Brownie**
- **Running unit tests with Foundry**
- **Running unit tests with Waffle**

- **Running unit tests with Remix**
- **Running unit tests with Ape**
- **Running unit tests with Hardhat**

Integration testing {#integration-testing-for-smart-contracts}

While unit testing debugs contract functions in isolation, integration tests evaluate the components of a smart contract as a whole. Integration testing can detect issues arising from cross-contract calls or interactions between different functions in the same smart contract. For example, integration tests can help check if things like inheritance and dependency injection work properly.

Integration testing is useful if your contract adopts a modular architecture or interfaces with other on-chain contracts during execution. One way of running integration tests is to fork the blockchain at a specific height (using a tool likeForge or Hardhat and simulate interactions between your contract and deployed contracts.

The forked blockchain will behave similarly to Mainnet and have accounts with associated states and balances. But it only acts as a sandboxed local development environment, meaning you won't need real ETH for transactions, for example, nor will your changes affect the real Ethereum protocol.

Property-based testing {#property-based-testing-for-smart-contracts}

Property-based testing is the process of checking that a smart contract satisfies some defined property. Properties assert facts about a contract's behavior that are expected to remain true in different scenarios—an example of a smart contract property could be "Arithmetic operations in the contract never overflow or underflow."

Static analysis and **dynamic analysis** are two common techniques for executing property-based testing, and both can verify that the code for a program (a smart contract in this case) satisfies some predefined property. Some property-based testing tools come with predefined rules about expected contract properties and check the code against those rules, while others allow you to create custom properties for a smart contract.

Static analysis {#static-analysis}

A static analyzer takes as input the source code of a smart contract and outputs results declaring whether a contract satisfies a property or not. Unlike dynamic analysis, static analysis doesn't involve executing a contract to analyze it for correctness. Static analysis instead reasons about all the possible paths that a smart contract could take during execution (i.e., by examining the structure of the source code to determine what it would mean for the contracts operation at runtime).

Linting and static testing are common methods for running static analysis on contracts. Both require analyzing low-level representations of a contracts execution such as abstract syntax trees and control flow graphs output by the compiler.

In most cases, static analysis is useful for detecting safety issues like use of unsafe constructs, syntax errors, or violations of coding standards in a contracts code. However, static analyzers are known to be generally unsound at detecting deeper vulnerabilities, and may produce excessive false positives.

Dynamic analysis {#dynamic-analysis}

Dynamic analysis generates symbolic inputs (e.g., insymbolic execution) or concrete inputs (e.g., infuzzing) to a smart contracts functions to see if any execution trace(s) violates specific properties. This form of property-based testing differs from unit tests in that test cases cover multiple scenarios and a program handles the generation of test cases.

Fuzzing is an example of a dynamic analysis technique for verifying arbitrary properties in smart contracts. A fuzzer invokes functions in a target contract with random or malformed variations of a defined input value. If the smart contract enters an error state (e.g., one where an assertion fails), the problem is flagged and inputs that drive execution toward the vulnerable path are produced in a report.

Fuzzing is useful for evaluating a smart contracts input validation mechanism since improper handling of unexpected inputs might result in unintended execution and produce dangerous effects. This form of property-based testing can be ideal for many reasons:

1. **Writing test cases to cover many scenarios is difficult.**A property test only requires that you define a behavior and a range of data to test the behavior with—the program automatically generates test cases based on the defined property.
2. **Your test suite may not sufficiently cover all possible paths within the program.**Even with 100% coverage, it is possible to miss out on edge cases.
3. **Unit tests prove a contract executes correctly for sample data, but whether the contract executes correctly for inputs outside the sample remains unknown.** Property tests execute a target contract with multiple variations of a given input value to find execution traces that cause assertion failures. Thus, a property test provides more guarantees that a contract executes correctly for a broad class of input data.

Guidelines for running property-based testing for smart contracts {#running-property-based-tests}

Running property-based testing typically starts with defining a property (e.g., absence ofinteger overflows) or collection of properties that you want to verify in a smart contract. You may also need to define a range of values within which the program can generate data for transaction inputs when writing property tests.

Once configured properly, the property testing tool will execute your smart contracts functions with randomly generated inputs. If there are any assertion violations, you should get a report with concrete input data that violates the property under evaluation. See some of the guides below to get started with running property-based testing with different tools:

- **Static analysis of smart contracts with Slither**
- **Property-based testing with Brownie**
- **Fuzzing contracts with Foundry**
- **Fuzzing contracts with Echidna**
- **Symbolic execution of smart contracts with Manticore**
- **Symbolic execution of smart contracts with Mythril**

Manual testing for smart contracts {#manual-testing-for-smart-contracts}

Manual testing of smart contracts often comes later in the development cycle after running automated tests. This form of testing evaluates the smart contract as one fully integrated product to see if it performs as specified in the technical requirements.

Testing contracts on a local blockchain {#testing-on-local-blockchain}

While automated testing performed in a local development environment can provide useful debugging information, you'll want to know how your smart contract behaves in a production environment. However, deploying to the main Ethereum chain incurs gas fees—not to mention that you or your users can lose real money if your smart contract still has bugs.

Testing your contract on a local blockchain (also known as adevelopment network) is a recommended alternative to testing on Mainnet. A local blockchain is a copy of the Ethereum blockchain running locally on your computer which simulates the behavior of Ethereum's execution layer. As such, you can program transactions to interact with a contract without incurring significant overhead.

Running contracts on a local blockchain could be useful as a form of manual integration testing.Smart contracts are highly composable, allowing you to integrate with existing protocols—but you'll still need to ensure that such complex on-chain interactions produce the correct results.

More on development networks.

Testing contracts on testnets {#testing-contracts-on-testnets}

A test network or testnet works exactly like Ethereum Mainnet, except that it uses Ether (ETH) with no real-world value. Deploying your contract on a testnet means anyone can interact with it (e.g., via the dapp's frontend) without putting funds at

risk.

This form of manual testing is useful for evaluating the end-to-end flow of your application from a user's point of view. Here, beta testers can also perform trial runs and report any issues with the contract's business logic and overall functionality.

Deploying on a testnet after testing on a local blockchain is ideal since the former is closer to the behavior of the Ethereum Virtual Machine. Therefore, it is common for many Ethereum-native projects to deploy dapps on testnets to evaluate a smart contracts operation under real-world conditions.

More on Ethereum testnets.

Testing vs. formal verification {#testing-vs-formal-verification}

While testing helps confirm that a contract returns the expected results for some data inputs, it cannot conclusively prove the same for inputs not used during tests. Testing a smart contract, therefore, cannot guarantee "functional correctness" (i.e., it cannot show that a program behaves as required for *all* sets of input values).

Formal verification is an approach to assessing the correctness of software by checking whether a formal model of the program matches the formal specification. A formal model is an abstract mathematical representation of a program, while a formal specification defines a program's properties (i.e., logical assertions about the program's execution).

Because properties are written in mathematical terms, it becomes possible to verify that a formal (mathematical) model of the system satisfies a specification using logical rules of inference. Thus, formal verification tools are said to produce 'mathematical proof' of a system's correctness.

Unlike testing, formal verification can be used to verify a smart contracts execution satisfies a formal specification for *all* executions (i.e., it has no bugs) without needing to execute it with sample data. Not only does this reduce time spent on running dozens of unit tests, but it is also more effective at catching hidden vulnerabilities. That said, formal verification techniques lie on a spectrum depending on their difficulty of implementation and usefulness.

More on formal verification for smart contracts.

Testing vs audits and bug bounties {#testing-vs-audits-bug-bounties}

As mentioned, rigorous testing can rarely guarantee the absence of bugs in a contract; formal verification approaches can provide stronger assurances of correctness but are currently difficult to use and incur considerable costs.

Still, you can further increase the possibility of catching contract vulnerabilities by getting an independent code review. Smart contract audits and bug bounties are two ways of getting others to analyze your contracts.

Audits are performed by auditors experienced at finding cases of security flaws and poor development practices in smart contracts. An audit will usually include testing (and possibly formal verification) as well as a manual review of the entire codebase.

Conversely, a bug bounty program usually involves involves offering a financial reward to an individual (commonly described as whitehat hackers) that discovers a vulnerability in a smart contract and discloses it to developers. Bug bounties are similar to audits since it involves asking others to help find defects in smart contracts.

The major difference is that bug bounty programs are open to the wider developer/hacker community and attract a broad class of ethical hackers and independent security professionals with unique skills and experience. This may be an advantage over smart contract audits that mainly rely on teams who may possess limited or narrow expertise.

Testing tools and libraries {#testing-tools-and-libraries}

Unit testing tools {#unit-testing-tools}

- **solidity-coverage** - *Code coverage tool for smart contracts written in Solidity.*

- **Waffle** - *Framework for advanced smart contract development and testing (based on ethers.js).*
- **Remix Tests** - *Tool for testing Solidity smart contracts. Works underneath Remix IDE "Solidity Unit Testing" plugin which is used to write and run test cases for a contract.*
- **OpenZeppelin Test Helpers** - *Assertion library for Ethereum smart contract testing. Make sure your contracts behave as expected!*
- **Brownie unit testing framework** - *Brownie utilizes Pytest, a feature-rich test framework that lets you write small tests with minimal code, scales well for large projects, and is highly extendable.*
- **Foundry Tests** - *Foundry offers Forge, a fast and flexible Ethereum testing framework capable of executing simple unit tests, gas optimization checks, and contract fuzzing.*
- **Hardhat Tests** - *Framework for testing smart contracts based on ethers.js, Mocha, and Chai.*
- **ApeWorx** - *Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine.*

Property-based testing tools {#property-based-testing-tools}

Static analysis tools {#static-analysis-tools}

- **Slither** - *Python-based Solidity static analysis framework for finding vulnerabilities, enhancing code comprehension, and writing custom analyses for smart contracts.*
- **Ethlint** - *Linter for enforcing style and security best practices for the Solidity smart contract programming language.*

Dynamic analysis tools {#dynamic-analysis-tools}

- **Echidna** - *Fast contract fuzzer for detecting vulnerabilities in smart contracts through property-based testing.*
- **Diligence Fuzzing** - *Automated fuzzing tool useful for detecting property violations in smart contract code.*
- **Manticore** - *Dynamic symbolic execution framework for analyzing EVM bytecode.*
- **Mythril** - *EVM bytecode assessment tool for detecting contract vulnerabilities using taint analysis, concolic analysis, and control flow checking.*
- **Diligence Scribble** - *Scribble is a specification language and runtime verification tool that allows you to annotate smart contracts with properties that allow you to automatically test the contracts with tools such as Diligence Fuzzing or MythX.*

Related tutorials {#related-tutorials}

- An overview and comparison of different testing products _
- How to use Echidna to test smart contracts
- How to use Manticore to find smart contract bugs
- How to use Slither to find smart contract bugs
- How to mock Solidity contracts for testing
- How to run unit tests in Solidity using Foundry

Further reading {#further-reading}

- An in-depth guide to testing Ethereum smart contracts
- How to test ethereum smart contracts
- MolochDAO's unit testing guide for developers
- How to test smart contracts like a rockstar