

Lesson 14 - Formal Verification / Symbolic Execution and Advanced Testing

Formal Verification

"Program testing can be used very effectively to show the presence of bugs but never to show their absence." — Edsger Dijkstra

See overview [resource](#)

Introduction

Formal Verification is the process by which one proves properties of a system mathematically. In order to do that, one writes a formal specification of the application behavior. The formal specification is analogous to a Statement of Intended Behavior, but it is written in a machine-readable language.

The formal specification is later proved or disproved using one of the available tools.

The correctness verification is about respecting the specifications that determine how users can interact with the smart contracts and how the smart contracts should behave when used correctly.

There are two approaches used to verify the correctness:

- the formal verification and
- the programming correctness.

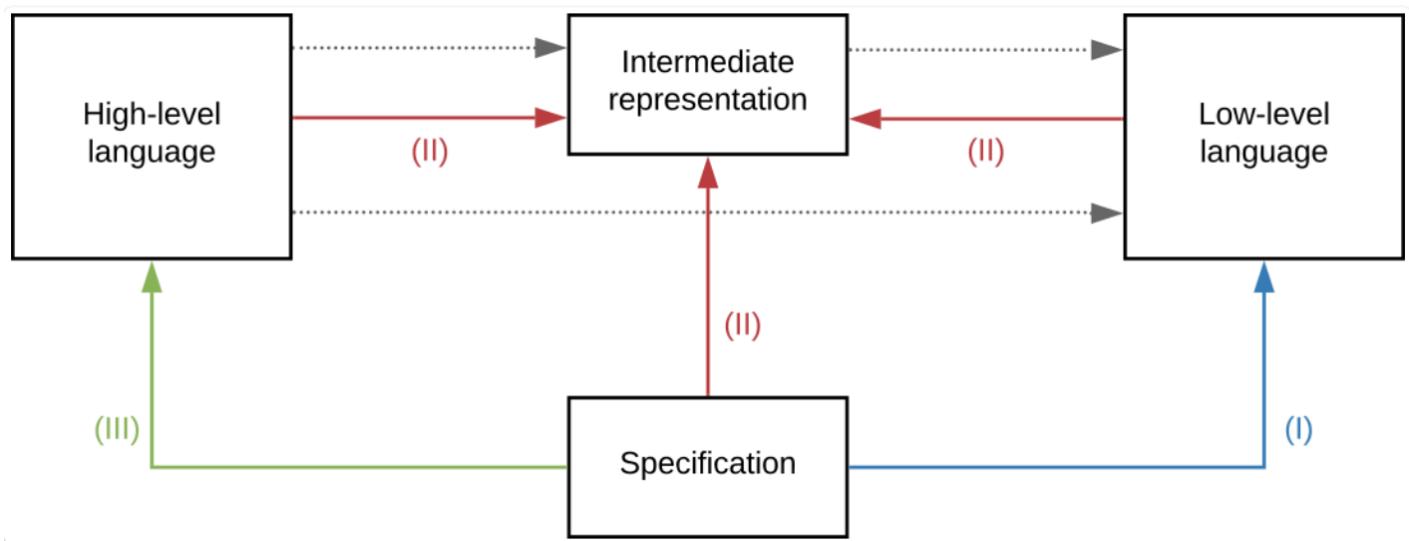
The formal verification methods are based on formal methods (mathematical methods), while the programming correctness methods are based on ensuring the programming as code is correct, which means the program runs without entering an infinite loop and gives correct outputs for correct inputs.

In the case of smart contracts verification, we can improve smart contracts security by ensuring the correctness of contracts using formal verification.

We may distinguish three major verification approaches :

1. a specification may be assessed directly at bytecode level. As contract sources are not necessarily available, this approach provides a way to assess some properties on already deployed contracts.
2. intermediate representations may be specifically designed as targets for verification tools such as proof assistants. This can offer a very suitable environment for code optimisation and dynamic verification according to specifications. The intermediate code representation may come both from compilation of a high-level contract or decompilation of low-level code.
3. some tools also reason directly on high-level languages. This approach offers a precious direct feedback to developers at verification time.

Different methods of verification based on the comparison object with the specification:



Contrary to Tezos and Cardano for instance, a [formal semantics of the EVM](#) was only described ex-post. And as the Solidity compiler changes rapidly, in the absence of formal semantics of the language that would allow correct-by-construction automatic generation of verification tools, the latter would need to follow the rate of change as well. These reasons make formal verification of smart contracts way harder on Ethereum than on other blockchains despite tremendous work initiated by several teams.

Two essential notions must be kept in mind to understand the challenge of formally verifying smart contracts :

1. **Verification can be thought of as testing a set of properties** (not all, only the ones we formalize) for *all* possible scenarios (including not only parameters but also message senders, blockchain state, storage, etc.). We don't prove smart contracts, we prove some of their properties by proving their correctness according to a specification.
2. **Proven correctness of all translations determines the level of confidence one can have in the entire framework.** If formal verification of a program is performed on its intermediate representation, a backwards translation will allow for meaningful messages to be displayed in the higher-level original language. Furthermore, if translation to machine bytecode is not secure, then no one can formally trust its execution, regardless of the verification effort at other levels. To be considered valid, a proof must be generated within a single, trusted logical framework, from the level of the specification language to the virtual machine execution level.

As an example of functional specifications, some properties of interest for an ERC20 implementation can be:

1. a *function level contract* stating that the function `transfer` decreases the balance of the sender in a determined `amount` and increases the destination balance in the same `amount` without affecting other balances. The sender must have enough tokens to perform this operation.
 2. a *contract level invariant* prescribing that the sum of balances is always equal to the total supply.
 3. a *temporal property* specifying a property that must hold for a sequence of transactions to be valid, e.g., `totalSupply` does not increase unless the `mint` function is invoked.
-

Pros and cons of formal verification

Formal verification of smart contracts isn't easy. There's a steep learning curve and a high entry threshold. A developer needs to undergo special training to be able to formalize requirements using formal verification tools.

Additionally, formal verification requires segregation of duties, in order to be effective. If the writer of the specifications is the coder of the application, the effectiveness of formal verification is greatly reduced.

Pros

- It doesn't rely on compilers, this allows formal verification to catch bugs that were introduced during compilation or other transformations of the source code.
- It's language-independent, you can easily verify smart contracts written in Solidity, Viper, or other languages that may appear in the future.
- A formal specification can work as documentation for the contract itself.

Cons

- It requires a specially prepared Ethereum execution environment.
 - A formal specification of the contract itself is required. Creating a formal specification is a long and difficult process and demands a lot of preparation from your development team. Specifying a program involves abstracting its properties and is thus a difficult task.
 - If there are any errors in the specification, they will be left undetected, every false requirement will be perceived as correct. Therefore, the verification won't detect a problem with the smart contract.
-

Formal Verification versus Symbolic Execution

Symbolic execution is a dynamic analysis technique that aims to explore different execution paths of a program by using symbolic values instead of concrete inputs. It involves tracking symbolic representations of program inputs and performing operations on these symbols to reason about the program's behavior. During symbolic execution, the program's paths are explored symbolically, and constraints are collected along each path. These constraints represent the conditions under which a particular path is taken.

The main advantage of symbolic execution is that it can systematically explore various execution paths, even those that are difficult to reach through traditional testing methods. By reasoning symbolically about program inputs and constraints, it can uncover bugs, verify properties, and generate test cases that satisfy specific conditions.

Formal verification, on the other hand, is a static analysis technique that aims to mathematically prove the correctness of a program with respect to a given specification or set of properties. It involves creating a formal model of the program and its intended behavior, often expressed in a formal language or logic. The formal model captures the program's semantics, and using formal methods, properties about the program can be formally stated and verified.

Formal verification typically involves techniques such as theorem proving, model checking, or abstract interpretation. These methods use rigorous mathematical reasoning to verify properties like safety, liveness, or functional correctness of the program. Formal verification can provide strong guarantees about program correctness, but it can also be computationally expensive and require significant expertise in formal methods.

Formal verification vs Unit Testing

Unit testing is usually cheaper than any other type of audit, as it's performed when developing a smart contract. Proper unit tests should reflect the specification and cover the smart contract with the help of use cases and functionality the specification describes. However, unit tests are just as imperfect as informal specifications. Even if the smart contract code is fully covered with unit tests, the developer may miss some edge cases that could lead to bugs or even security vulnerabilities like overflows or unprotected functions.

Formal verification vs Code Audits

Formal verification is not a silver bullet, or a substitute for a good audit. Also in other industries where audits are conducted, they include not only the code base, but also the formal verification specification itself. Flaws in this specification will mean that some of the properties of the application are not actually proven in the end, with bugs possibly going unnoticed.

While no smart contract can be guaranteed as safe and free of bugs, a thorough code audit and formal verification process from a reputable security firm helps uncover critical, high severity bugs that otherwise could result in financial harm to users.

Formal verification vs Static Analysis Tools

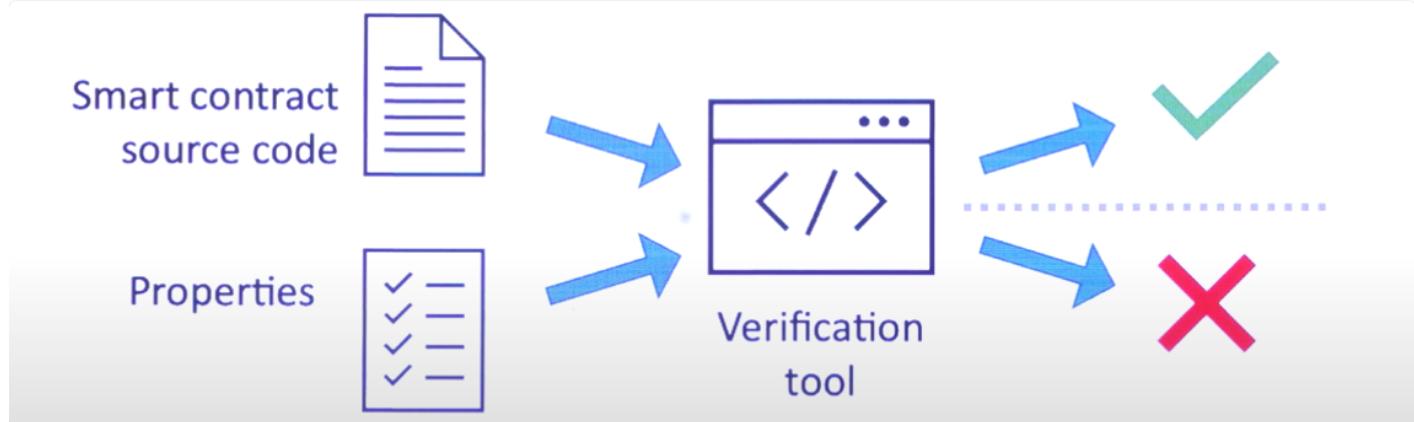
Static analysis tools are used to achieve the same goal as formal verification. During static analysis, a dedicated program (the static analyser) scans the smart contract or its bytecode in order to understand its behaviour and check for unexpected cases such as overflows and reentrancy vulnerabilities.

However, static analysers are limited in the range of vulnerabilities they can detect. In a sense, a static analyser attempts to perform the same formal verification with a one-fits-all specification that simply states, a smart contract shouldn't have any known vulnerabilities. Obviously, a custom specification is much better, as it will detect each and every possible vulnerability within a given smart contract.

Symbolic Execution

Useful [article](#)

The usual process in formal verification requires a set of properties to be supplied to the verification tool along with the code under test.



A common property for a token is the balance invariance, that is
The sum of the balances in the ERC20 contract is equal to the total supply.

Rather than having to supply the properties, we might want to use existing invariants in our code.

Imagine we have a function like this

```
function example(uint256 x ) public pure{
    uint256 y = x - 1;
    uint256 z = 1;

    if (y==12345){
        z = 0;
        // bad situation
    }
    else{
        z = 1;
    }
    assert (z!=0);
}
```

we could fuzz test this in Foundry, but it is unlikely to catch the value that would cause the assert to fail.

if we use symbolic execution with this function, we start by saying that x could take any value (for that datatype), then as we go through the code we can narrow down what values would be permissible.

so for us it is obvious that the assert will fail if x = 12345

To generalise and automate this we need an SMT solver such as z3, which can take in the above code and tell us that the assert will fail if x = 12345

We can have more complex example where we also have state that could be set in other transactions.

Taint Analysis

This is similar to symbolic execution but is used to track the flow of tainted data through a program.

Taint analysis focuses on identifying and tracking variables or data that can be influenced by untrusted or potentially malicious sources (hence "tainted" data).

The analysis marks the origin of tainted data and propagates the taint through the program to identify how it flows and interacts with other variables or sensitive operations.

Formal Verification Tools

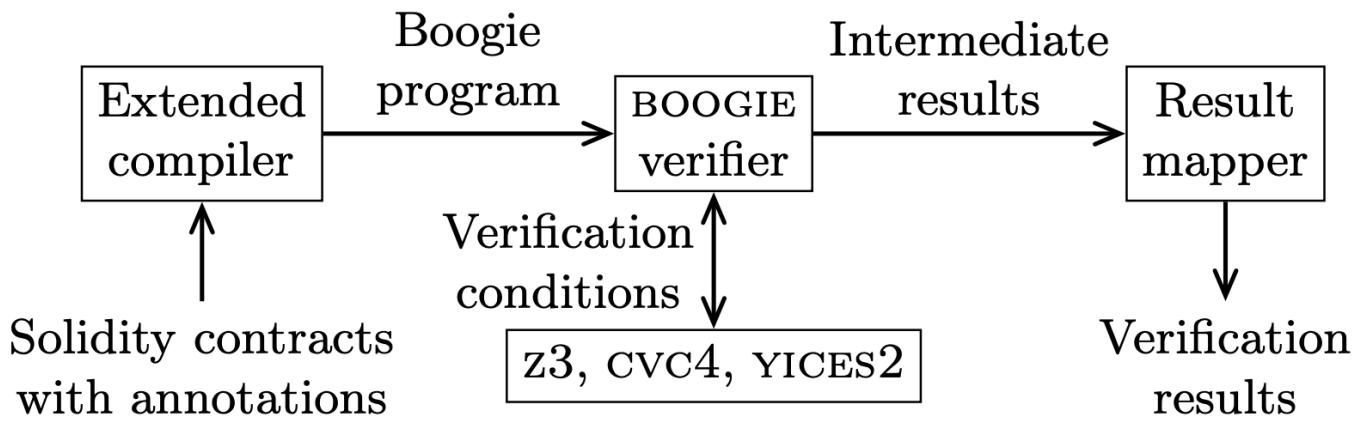
[solc-verify](#)

[Solc-verify](#) is a source-level formal verification tool for Solidity smart contracts, developed in collaboration with SRI International.

Solc-verify takes smart contracts written in Solidity and discharges verification conditions using modular program analysis and SMT solvers.

Built on top of the Solidity compiler, solc-verify reasons at the level of the contract source code. This enables solc-verify to effectively reason about high-level functional properties while modeling low-level language semantics (e.g., the memory model) precisely.

The contract properties, such as contract invariants, loop invariants, function pre- and post-conditions and fine grained access control can be provided as in-code annotations by the developer. This enables automated, yet user-friendly formal verification for smart contracts.



Overview of the solc-verify modules. The extended compiler creates a Boogie program from the Solidity contract, which is checked by the boogie verifier using SMT solvers. Finally, results are mapped back and presented at the Solidity code level.

VeriSol

[VeriSol](#) (Verifier for Solidity) is a Microsoft Research project for prototyping a formal verification and analysis system for smart contracts developed in the popular Solidity programming language. It is based on translating programs in Solidity language to programs in Boogie intermediate verification language, and then leveraging and extending the verification toolchain for Boogie programs. The following [blog](#) provides a high-level overview of the initial goals or VeriSol.

This tool takes contracts written in Solidity and tries to prove that the contract satisfies a set of given properties or provides a sequence of transactions that violates the properties.

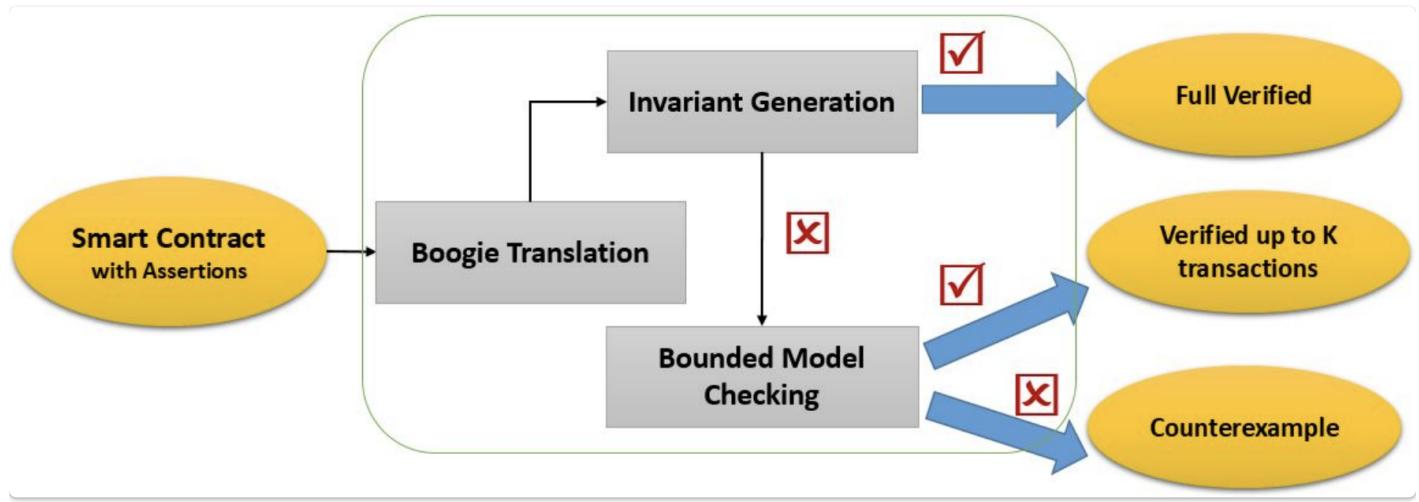
VeriSol directly understands `assert` and `require` clauses directly from Solidity but also includes a notion called Code Contracts (coined from [.NET Code Contracts](#)), where the language of contracts does not extend the language but uses a (dummy) set of additional (dummy) libraries that can be compiled by the Solidity compiler.

The function `transfer` equipped with assertions looks like this:

```
function transfer(address recipient, uint256 amount) public returns (bool) {  
    _transfer(msg.sender, recipient, amount);  
    assert (VeriSol.Old(_balances[msg.sender] + _balances[recipient]) ==  
    _balances[msg.sender] + _balances[recipient]);  
    assert (msg.sender == recipient || (_balances[msg.sender] ==  
    VeriSol.Old(_balances[msg.sender] - amount));  
  
    return true;  
}
```

The spec is straightforward. We expect the balance of the `recipient` to be increased by `amount`, and that amount of tokens should be decreased from the balance of `msg.sender`.

Schematic workflow of VERISOL:

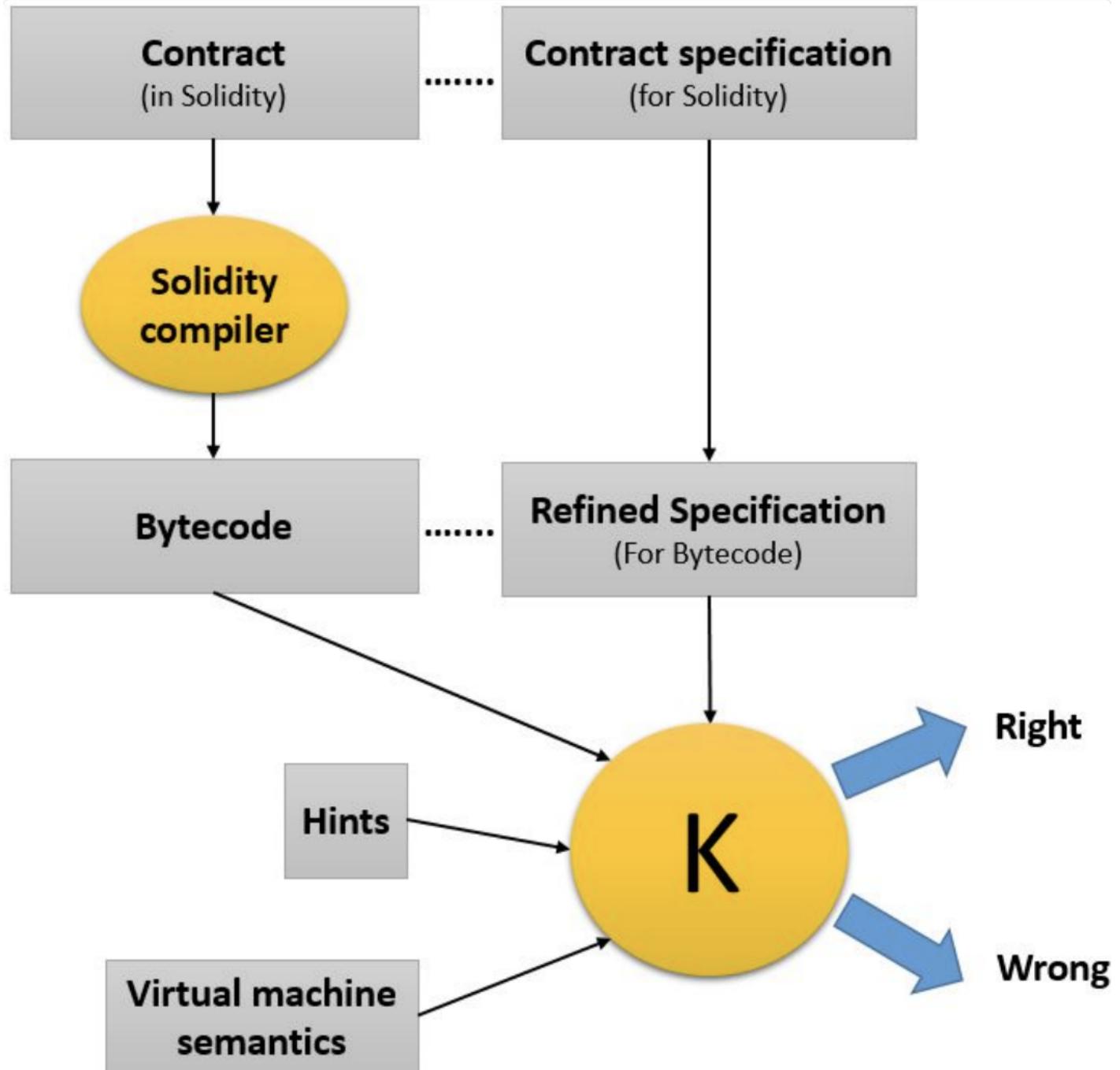


K Framework

The [K Framework](#) is one of the most robust and powerful language definition frameworks. It allows you to define your own programming language and provides you with a set of tools for that language, including both an executable model and a program verifier.

The K Framework provides a user-friendly, modular, and mathematically rigorous meta-language for defining programming languages, type systems, and analysis tools. K includes formal specifications for C, Java, JavaScript, PHP, Python, and Rust. Additionally, the K Framework enables verification of smart contracts.

The K-Framework is composed of 8 components listed in the following figure:



The [KEVM](#) provides the first machine-executable, mathematically formal, human readable and complete semantics for the EVM. The KEVM implements both the stack-based execution environment, with all of the EVM's opcodes, as well as the network's state, gas simulation, and even high-level aspects such as ABI call data.

If formal specifications of languages are defined, K Framework can handle automatic generation of various tools like interpreters and compilers. Nevertheless, this framework is very demanding (a lot of manual translations of specifications required, which are prone to error) and still suffer from some flaws (implementations of the EVM on the mainnet may not match the machine semantics for instance).

K Tutorial

This [tutorial](#) from Runtime Verification explains KEVM and its use with Foundry.

See the specification for [ERC20](#)

Solidity SMT Checker

See [documentation](#)

Also this useful [blog](#)

The SMTChecker module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements.

That is, it considers `require` statements as assumptions and tries to prove that the conditions inside `assert` statements are always true.

The other verification targets that the SMTChecker checks at compile time are:

- Arithmetic underflow and overflow.
- Division by zero.
- Trivial conditions and unreachable code.
- Popping an empty array.
- Out of bounds index access.
- Insufficient funds for a transfer.

To enable the SMTChecker, you must select [which engine should run](#), where the default is no engine. Selecting the engine enables the SMTChecker on all files.

The SMTChecker module implements two different reasoning engines, a Bounded Model Checker (BMC) and a system of Constrained Horn Clauses (CHC). Both engines are currently under development, and have different characteristics. The engines are independent and every property warning states from which engine it came. Note that all the examples above with counterexamples were reported by CHC, the more powerful engine.

By default both engines are used, where CHC runs first, and every property that was not proven is passed over to BMC. You can choose a specific engine via the CLI option `--model-checker-engine {all,bmc, chc, none}` or the JSON option `settings.modelChecker.engine= {all, bmc, chc, none}`

From the command line you can use

```
solc overflow.sol \  
--model-checker-targets "underflow,overflow" \  
--model-checker-engine all
```

language-bash

In Remix you can add

```
pragma experimental SMTChecker;
```

to your contract, though this is deprecated.

SMT in Foundry

Example taken from [Runtime verification](#)

For the following contract

```

contract ERC20 {
    address immutable owner;
    mapping(address => uint256) public balanceOf;

    constructor() {
        owner = msg.sender;
    }

    function mint(address user, uint256 amount) external {
        require(msg.sender == owner, "Only owner can mint");
        balanceOf[user] += amount;
    }

    function transfer(address to, uint amount) external {
        balanceOf[msg.sender] -= amount;
        balanceOf[to] += amount;
    }

    // Other functions...
}

```

We can write a test as follows

```

contract ERC20Test is Test {
    ERC20 token; // Contract under test

    address Alice = makeAddr("Alice");
    address Bob = makeAddr("Bob");
    address Eve = makeAddr("Eve");

    function setUp() public {
        token = new ERC20();
        token.mint(Alice, 10 ether);
        token.mint(Bob, 20 ether);
        token.mint(Eve, 30 ether);
    }

    function testTransfer(address from, address to, uint256 amount) public {
        vm.assume(token.balanceOf(from) >= amount);

        uint256 preBalanceFrom = token.balanceOf(from);
        uint256 preBalanceTo = token.balanceOf(to);

        vm.prank(from);
        token.transfer(to, amount);

        if(from == to) {

```

```
        assertEquals(token.balanceOf(from), preBalanceFrom);
        assertEquals(token.balanceOf(to), preBalanceTo);
    } else {
        assertEquals(token.balanceOf(from), preBalanceFrom - amount);
        assertEquals(token.balanceOf(to), preBalanceTo + amount);
    }
}
```

we can add the following lines to the `foundry.toml` file

```
[profile.default.model_checker]
contracts = {'../src/ERC20.sol' = ['ERC20']}
engine = 'all'
timeout = 10000
targets = ['assert']
```

Echidna

From their [documentation](#)

Echidna is a Haskell program designed for fuzzing/property-based testing of Ethereum smart contracts. It uses sophisticated grammar-based fuzzing campaigns based on a contract ABI to falsify user-defined predicates or Solidity assertions.

Features

- Generates inputs tailored to your actual code
- Optional corpus collection, mutation and coverage guidance to find deeper bugs
- Powered by [Slither](#) to extract useful information before the fuzzing campaign
- Source code integration to identify which lines are covered after the fuzzing campaign
- Curses-based retro UI, text-only or JSON output
- Automatic testcase minimization for quick triage
- Seamless integration into the development workflow
- Maximum gas usage reporting of the fuzzing campaign
- Support for a complex contract initialization with [Etheno](#) and Truffle

Echidna test runner

The core Echidna functionality is an executable called `echidna-test`.

This takes a contract and a list of invariants as input. 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. If it can't, you have some assurance the contract is safe.

You can integrate tests into github actions [workflow](#)

Invariants are expressed as Solidity functions with names that begin with `echidna_`, have no arguments, and return a boolean. For example, if you have some `balance` variable that should never go below `20`, you can write an extra function in your contract like this one:

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

To check these invariants, run:

```
$ echidna-test myContract.sol
```

See further [examples](#)

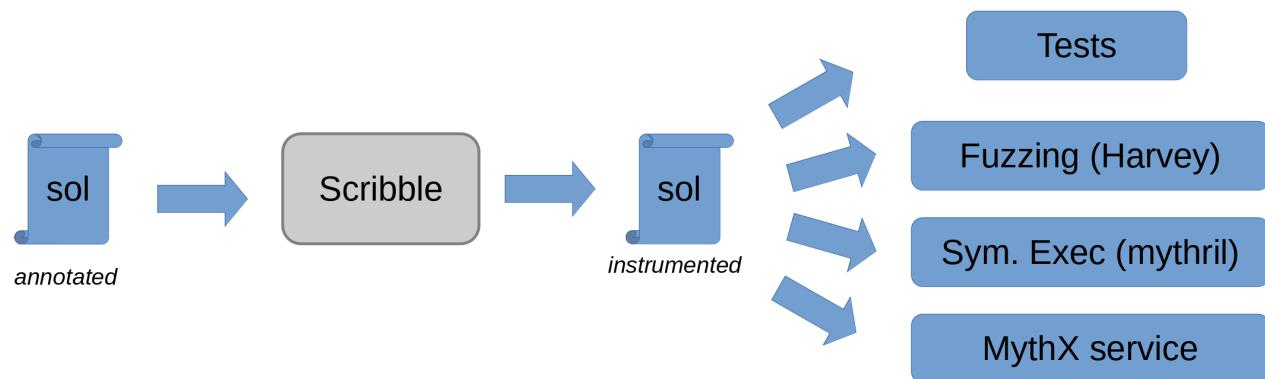
Introductory [course](#) on Echidna

Consensys Scribble

Scribble is a runtime verification tool for Solidity that transforms annotations in the [Scribble specification language](#) into concrete assertions that check the specification.

In other words, Scribble transforms existing contracts into contracts with equivalent behaviour, except that they also check properties.

With these *instrumented* contracts, you can use testing, fuzzing or symbolic execution (for example using Mythril or the MythX service) to check if your properties can be violated.



See [Documentation](#)

Installation

It is available via npm :

```
npm install -g eth-scribble
```

You add invariants to the code in the following format

```
import "Base.sol";  
  
contract Foo is Base {  
  
    /// #if_succeeds {:msg "P1"} y == x + 1;  
  
    function inc(uint x) public pure returns (uint y) {  
  
        return x+1;  
  
    }  
  
}
```

Scribble will then create instrumented files for you that can be used for testing.

Foundry

See notes from yesterday's lesson, Foundry is increasingly adding functionality in this area.

Certora

From their [documentation](#)

The Certora Prover is based on well-studied techniques from the formal verification community. **Specifications** define a set of rules that call into the contract under analysis and make various assertions about its behavior. Together with the contract under analysis, these rules are compiled to a logical formula called a **verification condition**, which is then proved or disproved by an SMT solver. If the rule is disproved, the solver also provides a concrete test case demonstrating the violation.

The Certora Prover verifies that a smart contract satisfies a set of rules written in a language called **Certora Verification Language (CVL)**. Each rule is checked on all possible inputs. Of course, this is not done by explicitly enumerating the inputs, but rather through symbolic techniques. Rules can check that a public contract method has the correct effects on the contract state or returns the correct value, etc... The syntax for expressing rules somewhat resembles Solidity, but also supports more features that are important for verification.

Example Rule

```
rule withdraw_succeeds {
    /* The env type represents the EVM parameters passed in every
    call (msg.* , tx.* , block.* variables in solidity      */
    env e;
    // Invoke function withdraw and assume it does not revert
    /* For non-envfree methods, the environment is passed as the first argument*/
    bool success = withdraw(e);
    assert success, "withdraw must succeed";
}
```

[Report for Aave Token V3](#)

Runtime Verification

What We Offer



Smart contract verification

Prove the correctness of smart contracts compiled for the Ethereum and Cardano virtual machines

[Choose your options](#)



NFT Checker

Mints can be gamed, tokens can be stolen, gas can be wasted, and bad code makes people wary of rug pulls.

[Let us help](#)



Protocol verification

Increase confidence in the correctness and security of the decentralized system powered by your protocol

[Understand the process](#)



Blockchain advisory services

Receive targeted advice from an RV senior engineer on the verification topic of your choice: smart contracts, tokens, protocols, or virtual machines

[Hire us!](#)



Firefly

Firefly is a set of quality assurance tools for Ethereum smart contracts aimed to help developers before the audit and formal verification stages

[Give it a try](#)



ERC20 token verifier

Check your token's full functional compliance with the ERC20 standard (the approve, transfer and transferFrom functions), before deploying it on mainnet

[Check your token](#)



Formal Verification

Like formal modeling, formal verification can also be applied to any system. For smart contracts, this package includes and further refines the Traditional Security Review ("Audit") and Formal Modeling packages, offering formal verification of the contract's EVM bytecode against its formal model/specification, thus eliminating any concerns of potential compiler errors. Specifically:

- Refine the contract's high-level formal specification to the bytecode-level to take into account the specifics of the underlying EVM.
- Formally verify the EVM bytecode of the contract against the refined formal specification. This package provides the ultimate formal guarantee for the correctness of your system or smart contract, and incorporates the best techniques and practices developed by the formal methods community.

30 - 90 days

[Ethereum 2.0 Deposit](#) | [Casper FFG](#) | [ERC20](#) | [Gnosis Safe](#)

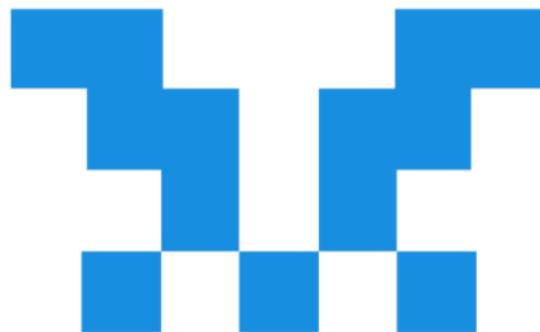
They use the K Framework and [KEVM](#)

See [ERC20-K: Formal Executable Specification of ERC20](#)

Resources

- [K Framework Semantics](#)
 - [The K Tutorial](#)
 - [K Framework Overview](#)
 - [Formal Verification article](#)
 - [Verification of smart contracts: A survey](#)
 - [Formal Verification of Smart Contracts with the K Framework](#)
 - [Solc-verify, a source-level formal verification tool for Solidity smart contracts](#)
-

Symbolic Execution Tools



Mythril

Mythril is a security analysis tool for EVM bytecode. It detects security vulnerabilities in EVM-compatible blockchains.

It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities. It's also used (in combination with other tools and techniques) in the [MythX](#) security analysis platform.

Manticore

[Manticore](#) is a dynamic symbolic execution tool, first described in the following blogposts by Trailofbits ([1](#), [2](#)).

Manticore performs the "heaviest weight" analysis. Like Echidna, Manticore verifies user-provided properties. It will need more time to run, but it can prove the validity of a property and will not report false alarms.

[Dynamic symbolic execution](#) (DSE) is a program analysis technique that explores a state space with a high degree of semantic awareness. This technique is based on the discovery of "program paths", represented as mathematical formulas called `path predicates`. Conceptually, this technique operates on path predicates in two steps:

1. They are constructed using constraints on the program's input.
2. They are used to generate program inputs that will cause the associated paths to execute.

This approach produces no false positives in the sense that all identified program states can be triggered during concrete execution. For example, if the analysis finds an integer overflow, it is guaranteed to be reproducible.



Halmos

See [Repo](#)

See [article](#)

Symbolic Bounded Model Checker for Ethereum Smart Contracts Bytecode

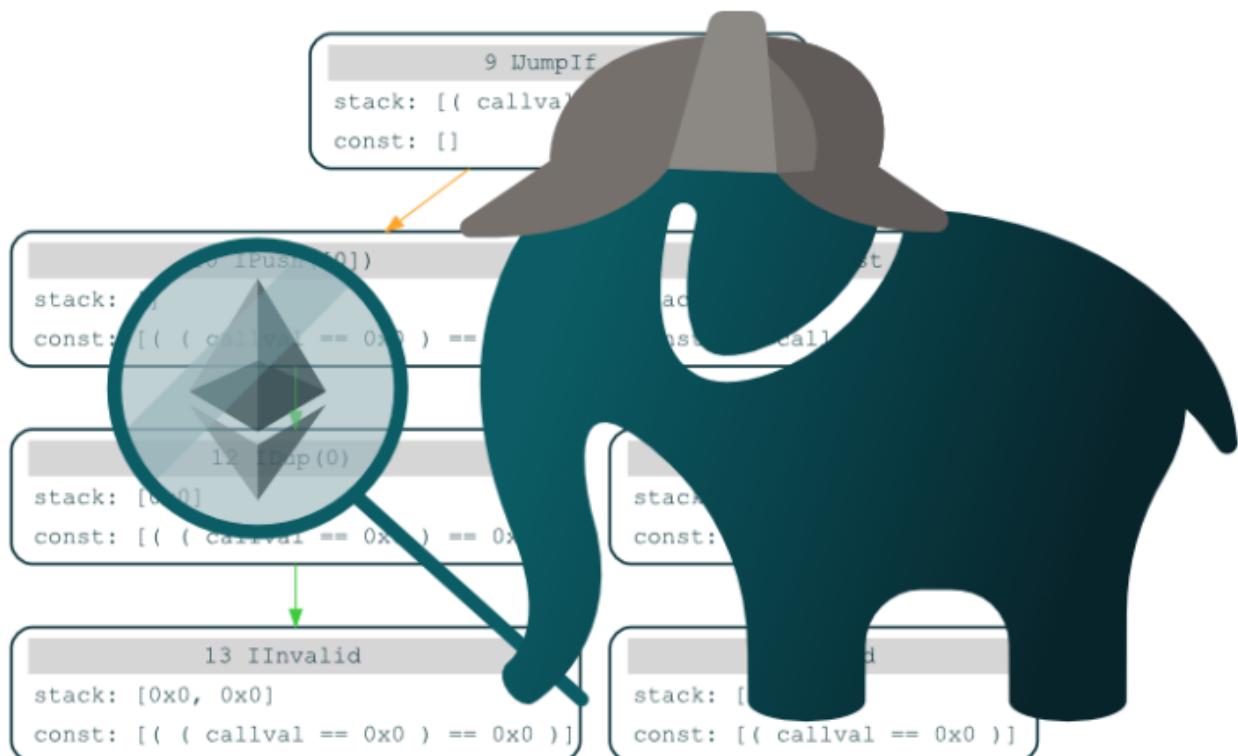
Halmos uses existing tests as formal specifications.

Halmos executes the given contract bytecode with symbolic function arguments and symbolic storage states, enabling it to systematically explore all possible behaviours of the contract.

EthBMC - A Bounded Model Checker for Smart Contracts

See [Repo](#)

See [Paper](#)



Introduction to Slither

See [repo](#)

Slither performs static analysis on source code, it has 'detectors' for known vulnerabilities / patterns.

It also includes some tools to help visualise / understand contracts and program flow.

Features

- Detects vulnerable Solidity code with low false positives (see the list of [trophies](#))
- Identifies where the error condition occurs in the source code
- Easily integrates into continuous integration and Truffle builds
- Built-in 'printers' quickly report crucial contract information
- Detector API to write custom analyses in Python
- Ability to analyze contracts written with Solidity ≥ 0.4
- Intermediate representation ([SlithIR](#)) enables simple, high-precision analyses
- Correctly parses 99.9% of all public Solidity code
- Average execution time of less than 1 second per contract

Slither can

1. produce markdown reports:
2. Display inheritance
3. produce call graphs
4. Show updatable variables and function authorisation
5. Check updatability
6. Flatten code
7. Read storage in a contract.

Installation

```
pip3 install slither-analyzer
```

Ethereum Security Toolbox

Trail of Bits have a security [toolbox](#) :

This includes

- [Echidna](#) property-based fuzz tester
- [Etheno](#) integration tool and differential tester
- [Manticore](#) symbolic analyzer and formal contract verifier
- [Slither](#) static analysis tool
- [Rattle](#) EVM lifter
- [Not So Smart Contracts](#) repository

Install in docker with

```
docker pull trailofbits/eth-security-toolbox
docker run -it trailofbits/eth-security-toolbox
```


ZIION from Halborn

See [Docs](#)

[Download](#)

ZIION BENEFITS



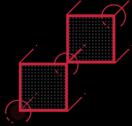
TURN-KEY VM

Get engineers ready faster by having a turnkey VM with all the best blockchain security tools needed, already compiled



OPENSOURCE

Community driven which means it's always free to use and open-source



COMPREHENSIVE

Reduce friction points by having a working and tested VM for each architecture and protocol family

ZIION FEATURES

ZIION contains over 100 tools, utilities, dependencies, and packages to immediately begin working with Solidity/EVM and Rust-based blockchains like:

Blockchains targeted

- Ethereum (EVM)
- Bitcoin
- Solana
- Polygon
- BSC
- Substrate
- Near

Tools include

Manticore, Slither, Mytril, Solgraph, Echidna, Certora-cli, Foundry

Testing

Fuzzing

Sometimes it's useful to test a function with many input variables at random in order to test edge-cases. This is called `fuzz testing`, or simply `fuzzing`.

An example of fuzzing using Foundry has been provided in the smart contract:

```
// Define the value(s) being fuzzed as an input argument      language-javascript
function test_FuzzValue(uint256 _value) public {
    // Define the boundaries for the fuzzing, in this case 0 and 99999
    _value = bound(_value, 0, 99999);
    // Call contract function with value
    a.store(_value);
    // Perform validation
    assertTrue(a.retrieve() == _value);
}
```

Interpreting results

Results may appear similar to this:

```
Running 4 tests for test/A.t.sol:ATest                                language-js
[PASS] test_FuzzValue(uint256) (runs: 256, μ: 31708, ~: 32330)
[PASS] test_GetValue() (gas: 7546)
[PASS] test_Log() (gas: 3930)
Logs:
here
0xb4c79dab8f259c7aee6e5b2aa729821864227e84
0x7109709ecfa91a80626ff3989d68f67f5b1dd12d
```

- "runs" refers to the amount of scenarios the fuzzer tested. By default, the fuzzer will generate 256 scenarios, however, this can be configured using the `FOUNDRY_FUZZ_RUNS` environment variable.
- "μ" (Greek letter mu) is the mean gas used across all fuzz runs.
- "˜" (tilde) is the median gas used across all fuzz runs.

Invariant Testing / Multi-step Fuzzing

An invariant is property that should always remain constant.

To test if the invariant can be broken it can be useful to make random calls to your contract. This is called `Invariant Testing`.

For example: Uniswap may want to test that `x*y` is always equal to `k` in their AMM contract. Or we may want to test that the contract owner never changes.

Invariant tests are prefixed with `invariant`.

```
// ETH can only be wrapped into WETH, WETH can only
// be unwrapped back into ETH. The sum of the Handler's
// ETH balance plus the WETH totalSupply() should always
// equal the total ETH_SUPPLY.
function invariant_conservationOfETH() public {
    assertEq(
        handler.ETH_SUPPLY(),
        address(handler).balance + weth.totalSupply()
    );
}
```

Recently Foundry invariant testing has improved a lot. It's very much worth taking the time to learn about it... Some good resources:

- Updated docs: <https://book.getfoundry.sh/forge/invariant-testing>
- Excellent tutorial: https://mirror.xyz/horsefacts.eth/Jex2YVaO65dda6zEyfM_-DXIXhOWCAoSpOx5PLocYgw

Differential Testing / Differential Fuzzing

Differential testing cross references multiple implementations of the same function by comparing each one's output.

Differential fuzzing programatically generates many values of x to find discrepancies and edge cases that manually chosen inputs might not reveal.

Some real life uses of this type of testing include:

- Comparing upgraded implementations to their predecessors
- Testing code against known reference implementations
- Confirming compatibility with third party tools and dependencies

Below we fuzz two different implementation of merkle trees to check for edge cases.

```
language-fs
import "openzeppelin-contracts/contracts/utils/cryptography/MerkleProof.sol";
//...
function testCompatibilityOpenZeppelinProver(bytes32[] memory _data, uint256
node) public {
    vm.assume(_data.length > 1);
    vm.assume(node < _data.length);
    bytes32 root = m.getRoot(_data);
    bytes32[] memory proof = m.getProof(_data, node);
    bytes32 valueToProve = _data[node];
    bool murkyVerified = m.verifyProof(root, proof, valueToProve);
    bool ozVerified = MerkleProof.verify(proof, root, valueToProve);
    assertTrue(murkyVerified == ozVerified);
}
```

See: <https://book.getfoundry.sh/forge/differential-ffi-testing>

Firefly testing tool

See [Docs](#)

Test runner. Firefly contains a reference implementation of the EVM equipped with a Web3 client so that it can be integrated into standard testing platforms for Ethereum. Firefly is a drop-in replacement for [ganache-cli](#), so users only need to replace their call to *ganache-cli* with a call to *firefly launch* when testing their contracts with [Truffle](#) or any other Web3-based testing platform.

The [Firefly](#) client is instrumented to collect bytecode-level coverage information when executing tests.

ERC20.sol

ERC20

Function not tested

```
37     function transferFrom(address sender, address recipient, uint256 amount) public returns (bool) {
38         _transfer(sender, recipient, amount);
39         _approve(sender, msg.sender, _allowances[sender][msg.sender] - amount);
40         return true;
41     }
42
43     function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
44         _approve(msg.sender, spender, _allowances[msg.sender][spender] + addedValue);
45         return true;
46     }
47
48     function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
49         _approve(msg.sender, spender, _allowances[msg.sender][spender] - subtractedValue);
50         return true;
51     }
52
53     function _transfer(address sender, address recipient, uint256 amount) internal {
54         (11/49) require(sender != address(0), "ERC20: transfer from the zero address");
55         (11/49) require(recipient != address(0), "ERC20: transfer to the zero address");
56
57         _balances[sender] = _balances[sender] - amount;
58         _balances[recipient] = _balances[recipient] + amount;
59         emit Transfer(sender, recipient, amount);
60     }
61
62     function _mint(address account, uint256 amount) internal {
63         require(account != address(0), "ERC20: mint to the zero address");
64
65         _balances[account] = _balances[account] + amount;
66         emit Transfer(address(0), account, amount);
67     }
68
69     function _burn(address account, uint256 amount) internal {
70         require(account != address(0), "ERC20: burn from the zero address");
71
72         _balances[account] = _balances[account] - amount;
73         emit Transfer(account, address(0), amount);
74     }
75
76     function _burnValue(uint256 value) internal {
77         require(value >= 1, "ERC20: burn value must be greater than zero");
78
79         _balances[address(0)] = _balances[address(0)] + value;
80         emit Transfer(address(0), address(0), value);
81     }
82
83     function _approve(address owner, address spender, uint256 amount) internal {
84         require(owner != address(0), "ERC20: approve from the zero address");
85         require(spender != address(0), "ERC20: approve to the zero address");
86
87         _allowances[owner][spender] = amount;
88         emit Approval(owner, spender, amount);
89     }
90
91     function _beforeTokenTransfer(address from, address to, uint256 amount) internal {
92         if (from == address(0)) {
93             require(to != address(0), "ERC20: transfer to the zero address");
94             _balances[from] = _balances[from] - amount;
95             _balances[to] = _balances[to] + amount;
96             emit Transfer(from, to, amount);
97         } else if (to == address(0)) {
98             _balances[from] = _balances[from] - amount;
99             emit Transfer(from, address(0), amount);
100        }
101    }
102
103    function _afterTokenTransfer(address from, address to, uint256 amount) internal {
104        if (from == address(0)) {
105            require(to != address(0), "ERC20: transfer to the zero address");
106            _balances[from] = _balances[from] - amount;
107            _balances[to] = _balances[to] + amount;
108            emit Transfer(from, to, amount);
109        } else if (to == address(0)) {
110            _balances[from] = _balances[from] - amount;
111            emit Transfer(from, address(0), amount);
112        }
113    }
114
115    function _checkForOverflows() internal {
116        require(_totalSupply <= type(uint256).max);
117    }
118
119    function _checkForUnderflows() internal {
120        require(_totalSupply >= 0);
121    }
122
123    function _checkForZeroAddress() internal {
124        require(_allowances[address(0)][address(0)] == 0);
125    }
126
127    function _checkForNonZeroAddress() internal {
128        require(_allowances[address(0)][address(0)] != 0);
129    }
130
131    function _checkForNonZeroBalance() internal {
132        require(_balances[address(0)] != 0);
133    }
134
135    function _checkForZeroBalance() internal {
136        require(_balances[address(0)] == 0);
137    }
138
139    function _checkForNonZeroTotalSupply() internal {
140        require(_totalSupply != 0);
141    }
142
143    function _checkForZeroTotalSupply() internal {
144        require(_totalSupply == 0);
145    }
146
147    function _checkForNonZeroAllowance() internal {
148        require(_allowances[address(0)][address(0)] != 0);
149    }
150
151    function _checkForZeroAllowance() internal {
152        require(_allowances[address(0)][address(0)] == 0);
153    }
154
155    function _checkForNonZeroBalanceOf() internal {
156        require(_balances[address(0)] != 0);
157    }
158
159    function _checkForZeroBalanceOf() internal {
160        require(_balances[address(0)] == 0);
161    }
162
163    function _checkForNonZeroTotalSupplyOf() internal {
164        require(_totalSupply != 0);
165    }
166
167    function _checkForZeroTotalSupplyOf() internal {
168        require(_totalSupply == 0);
169    }
170
171    function _checkForNonZeroAllowanceOf() internal {
172        require(_allowances[address(0)][address(0)] != 0);
173    }
174
175    function _checkForZeroAllowanceOf() internal {
176        require(_allowances[address(0)][address(0)] == 0);
177    }
178
179    function _checkForNonZeroBalanceOfOf() internal {
180        require(_balances[address(0)] != 0);
181    }
182
183    function _checkForZeroBalanceOfOf() internal {
184        require(_balances[address(0)] == 0);
185    }
186
187    function _checkForNonZeroTotalSupplyOfOf() internal {
188        require(_totalSupply != 0);
189    }
190
191    function _checkForZeroTotalSupplyOfOf() internal {
192        require(_totalSupply == 0);
193    }
194
195    function _checkForNonZeroAllowanceOfOf() internal {
196        require(_allowances[address(0)][address(0)] != 0);
197    }
198
199    function _checkForZeroAllowanceOfOf() internal {
200        require(_allowances[address(0)][address(0)] == 0);
201    }
202
203    function _checkForNonZeroBalanceOfOfOf() internal {
204        require(_balances[address(0)] != 0);
205    }
206
207    function _checkForZeroBalanceOfOfOf() internal {
208        require(_balances[address(0)] == 0);
209    }
210
211    function _checkForNonZeroTotalSupplyOfOfOf() internal {
212        require(_totalSupply != 0);
213    }
214
215    function _checkForZeroTotalSupplyOfOfOf() internal {
216        require(_totalSupply == 0);
217    }
218
219    function _checkForNonZeroAllowanceOfOfOf() internal {
220        require(_allowances[address(0)][address(0)] != 0);
221    }
222
223    function _checkForZeroAllowanceOfOfOf() internal {
224        require(_allowances[address(0)][address(0)] == 0);
225    }
226
227    function _checkForNonZeroBalanceOfOfOfOf() internal {
228        require(_balances[address(0)] != 0);
229    }
230
231    function _checkForZeroBalanceOfOfOfOf() internal {
232        require(_balances[address(0)] == 0);
233    }
234
235    function _checkForNonZeroTotalSupplyOfOfOfOf() internal {
236        require(_totalSupply != 0);
237    }
238
239    function _checkForZeroTotalSupplyOfOfOfOf() internal {
240        require(_totalSupply == 0);
241    }
242
243    function _checkForNonZeroAllowanceOfOfOfOf() internal {
244        require(_allowances[address(0)][address(0)] != 0);
245    }
246
247    function _checkForZeroAllowanceOfOfOfOf() internal {
248        require(_allowances[address(0)][address(0)] == 0);
249    }
250
251    function _checkForNonZeroBalanceOfOfOfOfOf() internal {
252        require(_balances[address(0)] != 0);
253    }
254
255    function _checkForZeroBalanceOfOfOfOfOf() internal {
256        require(_balances[address(0)] == 0);
257    }
258
259    function _checkForNonZeroTotalSupplyOfOfOfOfOf() internal {
260        require(_totalSupply != 0);
261    }
262
263    function _checkForZeroTotalSupplyOfOfOfOfOf() internal {
264        require(_totalSupply == 0);
265    }
266
267    function _checkForNonZeroAllowanceOfOfOfOfOf() internal {
268        require(_allowances[address(0)][address(0)] != 0);
269    }
270
271    function _checkForZeroAllowanceOfOfOfOfOf() internal {
272        require(_allowances[address(0)][address(0)] == 0);
273    }
274
275    function _checkForNonZeroBalanceOfOfOfOfOfOf() internal {
276        require(_balances[address(0)] != 0);
277    }
278
279    function _checkForZeroBalanceOfOfOfOfOfOf() internal {
280        require(_balances[address(0)] == 0);
281    }
282
283    function _checkForNonZeroTotalSupplyOfOfOfOfOfOf() internal {
284        require(_totalSupply != 0);
285    }
286
287    function _checkForZeroTotalSupplyOfOfOfOfOfOf() internal {
288        require(_totalSupply == 0);
289    }
290
291    function _checkForNonZeroAllowanceOfOfOfOfOfOf() internal {
292        require(_allowances[address(0)][address(0)] != 0);
293    }
294
295    function _checkForZeroAllowanceOfOfOfOfOfOf() internal {
296        require(_allowances[address(0)][address(0)] == 0);
297    }
298
299    function _checkForNonZeroBalanceOfOfOfOfOfOfOf() internal {
300        require(_balances[address(0)] != 0);
301    }
302
303    function _checkForZeroBalanceOfOfOfOfOfOfOf() internal {
304        require(_balances[address(0)] == 0);
305    }
306
307    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOf() internal {
308        require(_totalSupply != 0);
309    }
310
311    function _checkForZeroTotalSupplyOfOfOfOfOfOfOf() internal {
312        require(_totalSupply == 0);
313    }
314
315    function _checkForNonZeroAllowanceOfOfOfOfOfOfOf() internal {
316        require(_allowances[address(0)][address(0)] != 0);
317    }
318
319    function _checkForZeroAllowanceOfOfOfOfOfOfOf() internal {
320        require(_allowances[address(0)][address(0)] == 0);
321    }
322
323    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOf() internal {
324        require(_balances[address(0)] != 0);
325    }
326
327    function _checkForZeroBalanceOfOfOfOfOfOfOfOf() internal {
328        require(_balances[address(0)] == 0);
329    }
330
331    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOf() internal {
332        require(_totalSupply != 0);
333    }
334
335    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOf() internal {
336        require(_totalSupply == 0);
337    }
338
339    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOf() internal {
340        require(_allowances[address(0)][address(0)] != 0);
341    }
342
343    function _checkForZeroAllowanceOfOfOfOfOfOfOfOf() internal {
344        require(_allowances[address(0)][address(0)] == 0);
345    }
346
347    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOf() internal {
348        require(_balances[address(0)] != 0);
349    }
350
351    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOf() internal {
352        require(_balances[address(0)] == 0);
353    }
354
355    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOf() internal {
356        require(_totalSupply != 0);
357    }
358
359    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOf() internal {
360        require(_totalSupply == 0);
361    }
362
363    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOf() internal {
364        require(_allowances[address(0)][address(0)] != 0);
365    }
366
367    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOf() internal {
368        require(_allowances[address(0)][address(0)] == 0);
369    }
370
371    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOf() internal {
372        require(_balances[address(0)] != 0);
373    }
374
375    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOf() internal {
376        require(_balances[address(0)] == 0);
377    }
378
379    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOf() internal {
380        require(_totalSupply != 0);
381    }
382
383    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOf() internal {
384        require(_totalSupply == 0);
385    }
386
387    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOf() internal {
388        require(_allowances[address(0)][address(0)] != 0);
389    }
390
391    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOf() internal {
392        require(_allowances[address(0)][address(0)] == 0);
393    }
394
395    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOf() internal {
396        require(_balances[address(0)] != 0);
397    }
398
399    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOf() internal {
400        require(_balances[address(0)] == 0);
401    }
402
403    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOf() internal {
404        require(_totalSupply != 0);
405    }
406
407    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOf() internal {
408        require(_totalSupply == 0);
409    }
410
411    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOf() internal {
412        require(_allowances[address(0)][address(0)] != 0);
413    }
414
415    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOf() internal {
416        require(_allowances[address(0)][address(0)] == 0);
417    }
418
419    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOf() internal {
420        require(_balances[address(0)] != 0);
421    }
422
423    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOf() internal {
424        require(_balances[address(0)] == 0);
425    }
426
427    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOf() internal {
428        require(_totalSupply != 0);
429    }
430
431    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOf() internal {
432        require(_totalSupply == 0);
433    }
434
435    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOf() internal {
436        require(_allowances[address(0)][address(0)] != 0);
437    }
438
439    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOf() internal {
440        require(_allowances[address(0)][address(0)] == 0);
441    }
442
443    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
444        require(_balances[address(0)] != 0);
445    }
446
447    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
448        require(_balances[address(0)] == 0);
449    }
450
451    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
452        require(_totalSupply != 0);
453    }
454
455    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
456        require(_totalSupply == 0);
457    }
458
459    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
460        require(_allowances[address(0)][address(0)] != 0);
461    }
462
463    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
464        require(_allowances[address(0)][address(0)] == 0);
465    }
466
467    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
468        require(_balances[address(0)] != 0);
469    }
470
471    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
472        require(_balances[address(0)] == 0);
473    }
474
475    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
476        require(_totalSupply != 0);
477    }
478
479    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
480        require(_totalSupply == 0);
481    }
482
483    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
484        require(_allowances[address(0)][address(0)] != 0);
485    }
486
487    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
488        require(_allowances[address(0)][address(0)] == 0);
489    }
490
491    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
492        require(_balances[address(0)] != 0);
493    }
494
495    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
496        require(_balances[address(0)] == 0);
497    }
498
499    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
500        require(_totalSupply != 0);
501    }
502
503    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
504        require(_totalSupply == 0);
505    }
506
507    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
508        require(_allowances[address(0)][address(0)] != 0);
509    }
510
511    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
512        require(_allowances[address(0)][address(0)] == 0);
513    }
514
515    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
516        require(_balances[address(0)] != 0);
517    }
518
519    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
520        require(_balances[address(0)] == 0);
521    }
522
523    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
524        require(_totalSupply != 0);
525    }
526
527    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
528        require(_totalSupply == 0);
529    }
530
531    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
532        require(_allowances[address(0)][address(0)] != 0);
533    }
534
535    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
536        require(_allowances[address(0)][address(0)] == 0);
537    }
538
539    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
540        require(_balances[address(0)] != 0);
541    }
542
543    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
544        require(_balances[address(0)] == 0);
545    }
546
547    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
548        require(_totalSupply != 0);
549    }
550
551    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
552        require(_totalSupply == 0);
553    }
554
555    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
556        require(_allowances[address(0)][address(0)] != 0);
557    }
558
559    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
560        require(_allowances[address(0)][address(0)] == 0);
561    }
562
563    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
564        require(_balances[address(0)] != 0);
565    }
566
567    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
568        require(_balances[address(0)] == 0);
569    }
570
571    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
572        require(_totalSupply != 0);
573    }
574
575    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
576        require(_totalSupply == 0);
577    }
578
579    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
580        require(_allowances[address(0)][address(0)] != 0);
581    }
582
583    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
584        require(_allowances[address(0)][address(0)] == 0);
585    }
586
587    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
588        require(_balances[address(0)] != 0);
589    }
590
591    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
592        require(_balances[address(0)] == 0);
593    }
594
595    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
596        require(_totalSupply != 0);
597    }
598
599    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
600        require(_totalSupply == 0);
601    }
602
603    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
604        require(_allowances[address(0)][address(0)] != 0);
605    }
606
607    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
608        require(_allowances[address(0)][address(0)] == 0);
609    }
610
611    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
612        require(_balances[address(0)] != 0);
613    }
614
615    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
616        require(_balances[address(0)] == 0);
617    }
618
619    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
620        require(_totalSupply != 0);
621    }
622
623    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
624        require(_totalSupply == 0);
625    }
626
627    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
628        require(_allowances[address(0)][address(0)] != 0);
629    }
630
631    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
632        require(_allowances[address(0)][address(0)] == 0);
633    }
634
635    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
636        require(_balances[address(0)] != 0);
637    }
638
639    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
640        require(_balances[address(0)] == 0);
641    }
642
643    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
644        require(_totalSupply != 0);
645    }
646
647    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
648        require(_totalSupply == 0);
649    }
650
651    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
652        require(_allowances[address(0)][address(0)] != 0);
653    }
654
655    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
656        require(_allowances[address(0)][address(0)] == 0);
657    }
658
659    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
660        require(_balances[address(0)] != 0);
661    }
662
663    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
664        require(_balances[address(0)] == 0);
665    }
666
667    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
668        require(_totalSupply != 0);
669    }
670
671    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
672        require(_totalSupply == 0);
673    }
674
675    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
676        require(_allowances[address(0)][address(0)] != 0);
677    }
678
679    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
680        require(_allowances[address(0)][address(0)] == 0);
681    }
682
683    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
684        require(_balances[address(0)] != 0);
685    }
686
687    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
688        require(_balances[address(0)] == 0);
689    }
690
691    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
692        require(_totalSupply != 0);
693    }
694
695    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
696        require(_totalSupply == 0);
697    }
698
699    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
700        require(_allowances[address(0)][address(0)] != 0);
701    }
702
703    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
704        require(_allowances[address(0)][address(0)] == 0);
705    }
706
707    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
708        require(_balances[address(0)] != 0);
709    }
710
711    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
712        require(_balances[address(0)] == 0);
713    }
714
715    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
716        require(_totalSupply != 0);
717    }
718
719    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
720        require(_totalSupply == 0);
721    }
722
723    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
724        require(_allowances[address(0)][address(0)] != 0);
725    }
726
727    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
728        require(_allowances[address(0)][address(0)] == 0);
729    }
730
731    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
732        require(_balances[address(0)] != 0);
733    }
734
735    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
736        require(_balances[address(0)] == 0);
737    }
738
739    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
740        require(_totalSupply != 0);
741    }
742
743    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
744        require(_totalSupply == 0);
745    }
746
747    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
748        require(_allowances[address(0)][address(0)] != 0);
749    }
750
751    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
752        require(_allowances[address(0)][address(0)] == 0);
753    }
754
755    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
756        require(_balances[address(0)] != 0);
757    }
758
759    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
759        require(_balances[address(0)] == 0);
760    }
761
762    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
763        require(_totalSupply != 0);
764    }
765
766    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
767        require(_totalSupply == 0);
768    }
769
770    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
771        require(_allowances[address(0)][address(0)] != 0);
772    }
773
774    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
775        require(_allowances[address(0)][address(0)] == 0);
776    }
777
778    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
779        require(_balances[address(0)] != 0);
780    }
781
782    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
783        require(_balances[address(0)] == 0);
784    }
785
786    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
787        require(_totalSupply != 0);
788    }
789
790    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
791        require(_totalSupply == 0);
792    }
793
794    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
795        require(_allowances[address(0)][address(0)] != 0);
796    }
797
798    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
799        require(_allowances[address(0)][address(0)] == 0);
800    }
801
802    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
801        require(_balances[address(0)] != 0);
802    }
803
804    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
805        require(_balances[address(0)] == 0);
806    }
807
808    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
809        require(_totalSupply != 0);
810    }
811
812    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
813        require(_totalSupply == 0);
814    }
815
816    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
815        require(_allowances[address(0)][address(0)] != 0);
816    }
817
818    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
819        require(_allowances[address(0)][address(0)] == 0);
820    }
821
822    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
823        require(_balances[address(0)] != 0);
824    }
825
826    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
827        require(_balances[address(0)] == 0);
828    }
829
830    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
831        require(_totalSupply != 0);
832    }
833
834    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
835        require(_totalSupply == 0);
836    }
837
838    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
837        require(_allowances[address(0)][address(0)] != 0);
838    }
839
840    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
841        require(_allowances[address(0)][address(0)] == 0);
842    }
843
844    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
845        require(_balances[address(0)] != 0);
846    }
847
848    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
849        require(_balances[address(0)] == 0);
850    }
851
852    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
851        require(_totalSupply != 0);
852    }
853
854    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
853        require(_totalSupply == 0);
854    }
855
856    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
856        require(_allowances[address(0)][address(0)] != 0);
857    }
858
859    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
859        require(_allowances[address(0)][address(0)] == 0);
860    }
861
862    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
863        require(_balances[address(0)] != 0);
864    }
865
866    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
866        require(_balances[address(0)] == 0);
867    }
868
869    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
869        require(_totalSupply != 0);
870    }
871
872    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
872        require(_totalSupply == 0);
873    }
874
875    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
875        require(_allowances[address(0)][address(0)] != 0);
876    }
877
878    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
878        require(_allowances[address(0)][address(0)] == 0);
879    }
880
881    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
881        require(_balances[address(0)] != 0);
882    }
883
884    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
884        require(_balances[address(0)] == 0);
885    }
886
887    function _checkForNonZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
887        require(_totalSupply != 0);
888    }
889
890    function _checkForZeroTotalSupplyOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
890        require(_totalSupply == 0);
891    }
892
893    function _checkForNonZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
893        require(_allowances[address(0)][address(0)] != 0);
894    }
895
896    function _checkForZeroAllowanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
896        require(_allowances[address(0)][address(0)] == 0);
897    }
898
899    function _checkForNonZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
900        require(_balances[address(0)] != 0);
901    }
902
903    function _checkForZeroBalanceOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOfOf() internal {
903        require(_balances[address(0)] == 0
```