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.

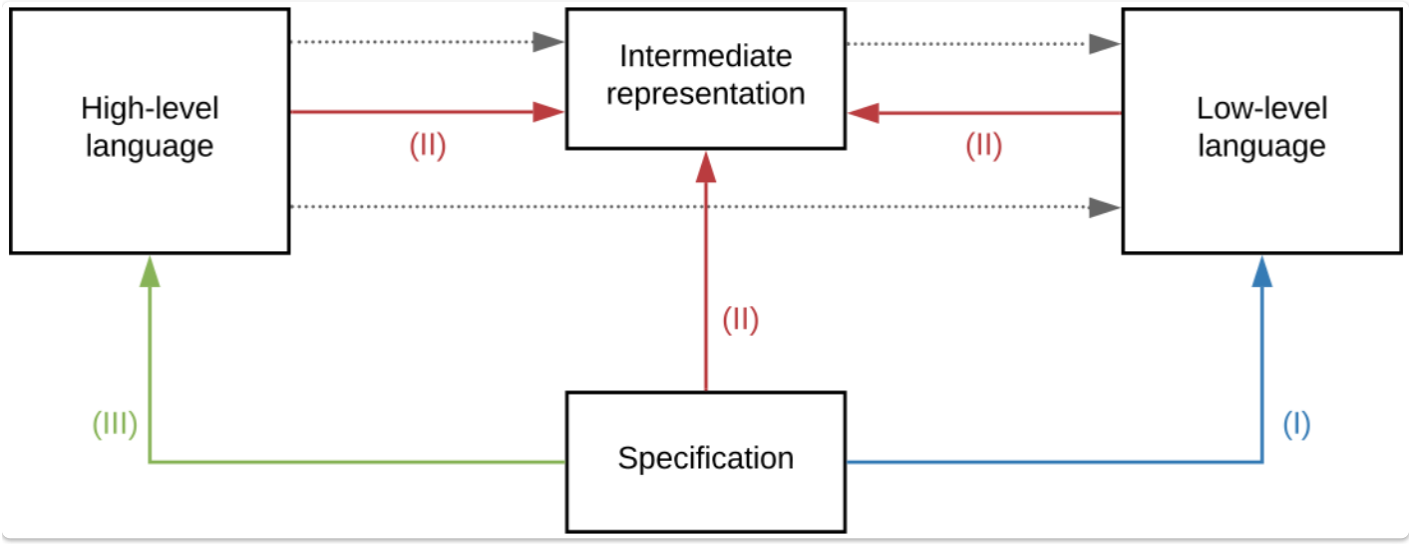
The formal verification methods are based on formal methods (mathematical methods), while the programming correctness methods are based on ensuring the programming as code is correct, which means the program runs without entering an infinite loop and gives correct outputs for correct inputs.

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

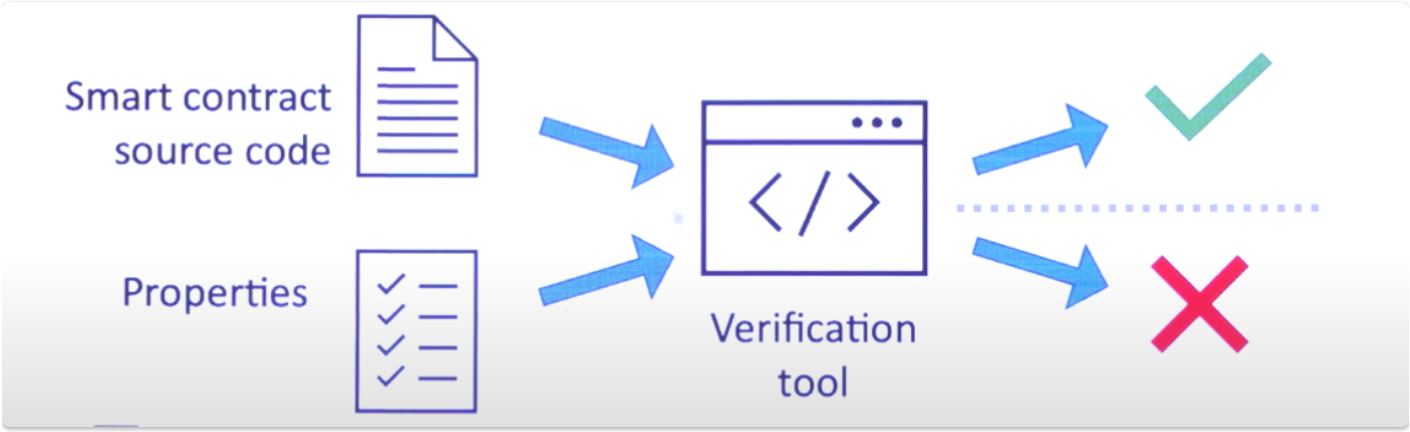
Static analysis tools are used to achieve the same goal as formal verification. During static analysis, a dedicated program (the static analyser) scans the smart contract or its bytecode in order to understand its behaviour and check for unexpected cases such as overflows and reentrancy vulnerabilities.

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.

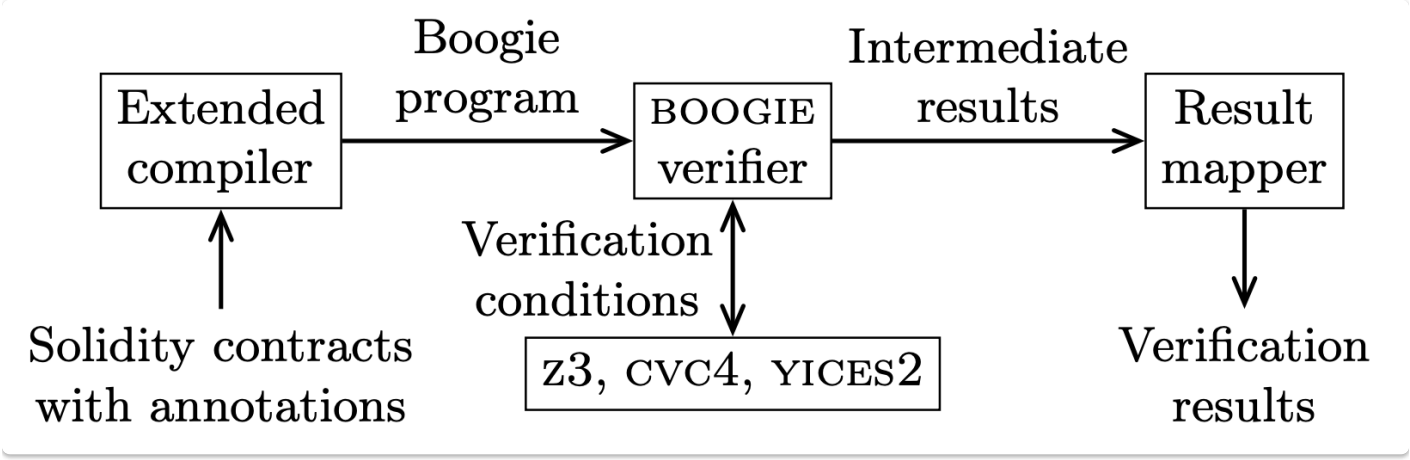
Taint Analysis

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 Soldity 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.

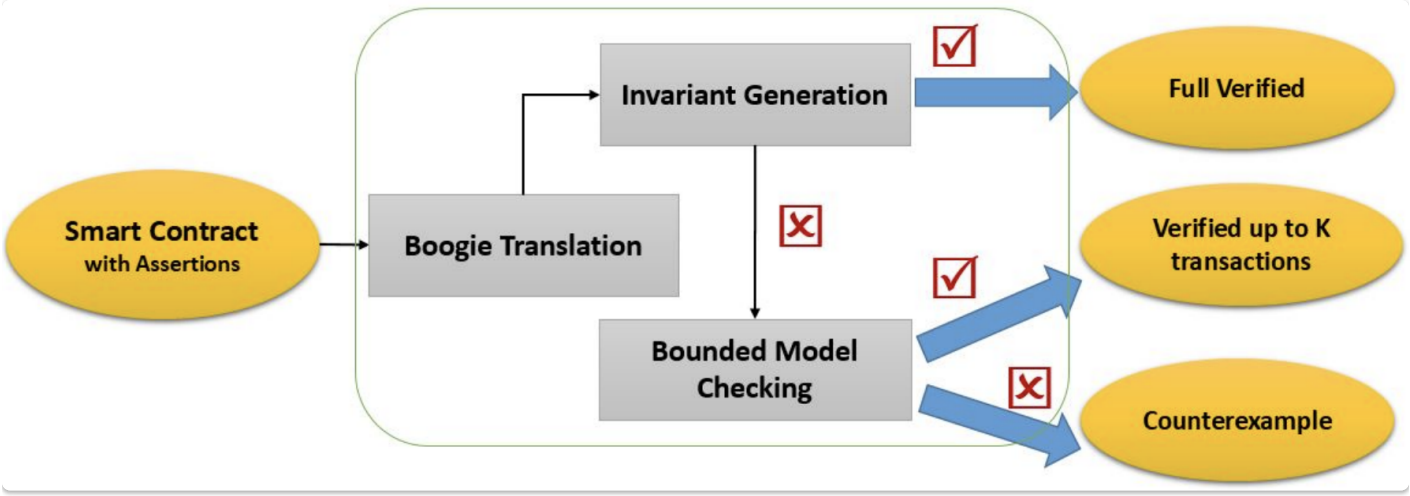
This tool takes contracts written in Solidity and tries to prove that the contract satisfies a set of given properties or provides a sequence of transactions that violates the properties.
VeriSol directly understands `assert` and `require` clauses directly from Solidity but also includes a notion called 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:

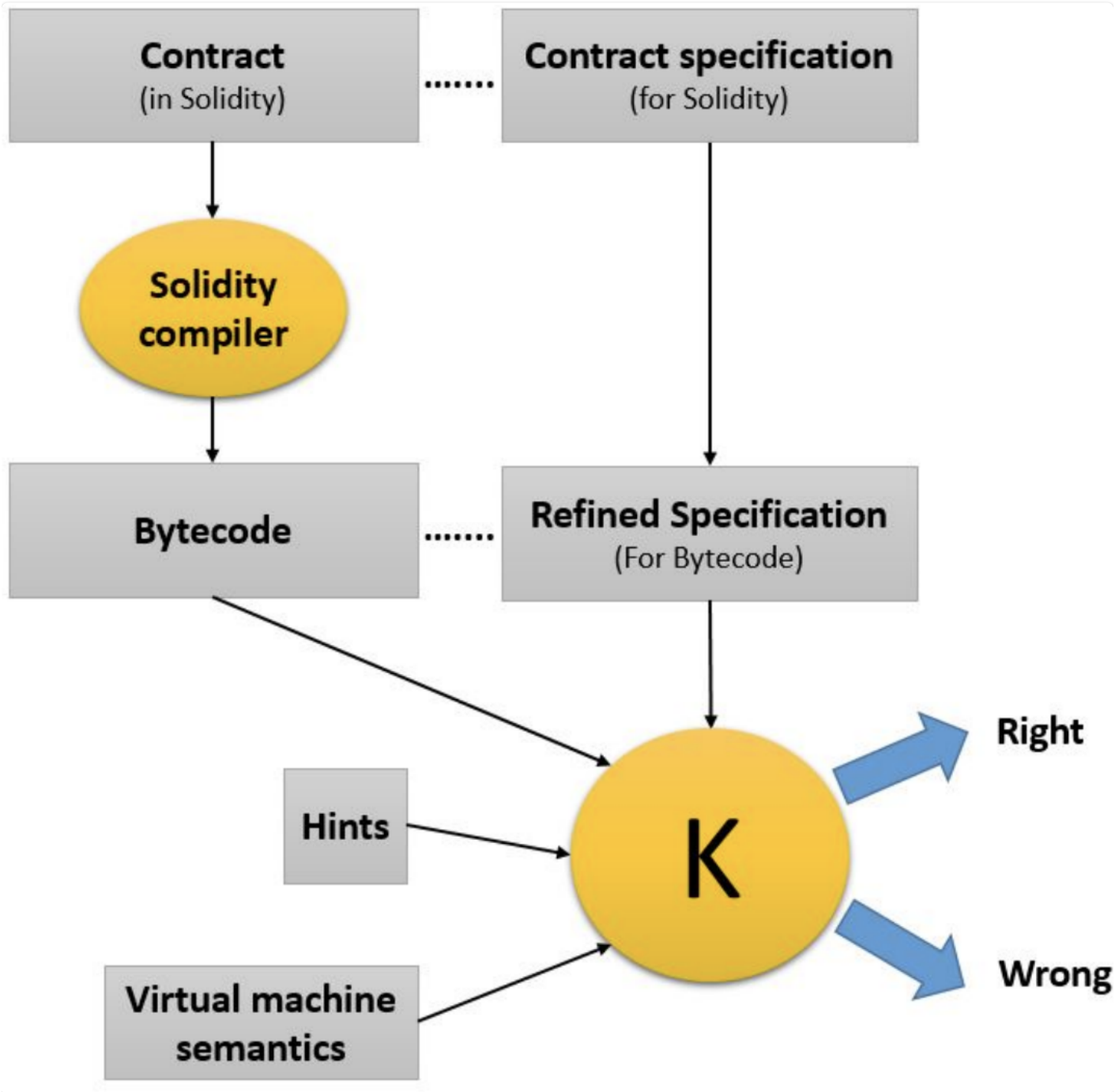

Smart Contract
with Assertions
Boogie Translation
Invariant Generation
Bounded Model
Checking
Full Verified
Verified up to K
transactions
Counterexample

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

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

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) {
            assertEq(token.balanceOf(from), preBalanceFrom);
            assertEq(token.balanceOf(to), preBalanceTo);
        } else {
            assertEq(token.balanceOf(from), preBalanceFrom -
amount);
            assertEq(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 smarts 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:

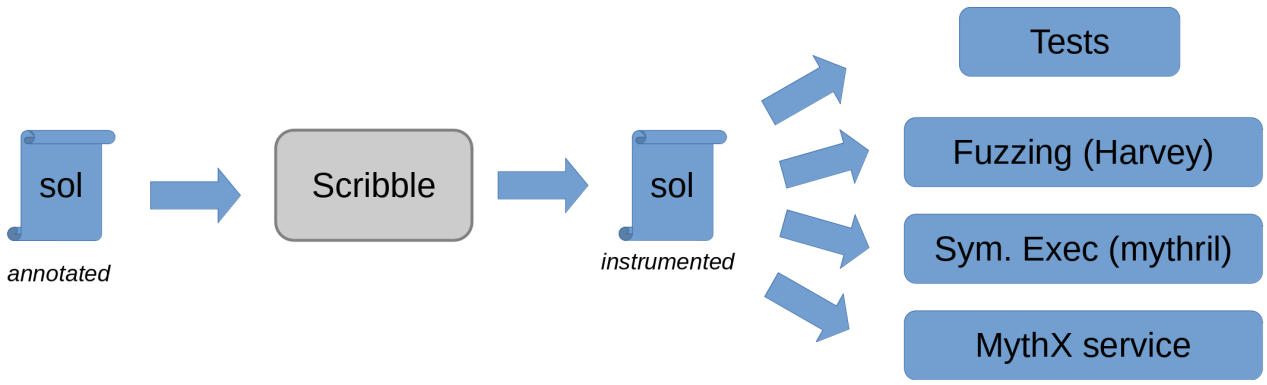
```
  function echidna_check_balance() public returns (bool) {
    return(balance >= 20);
}

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

30 - 90 days

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.

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 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

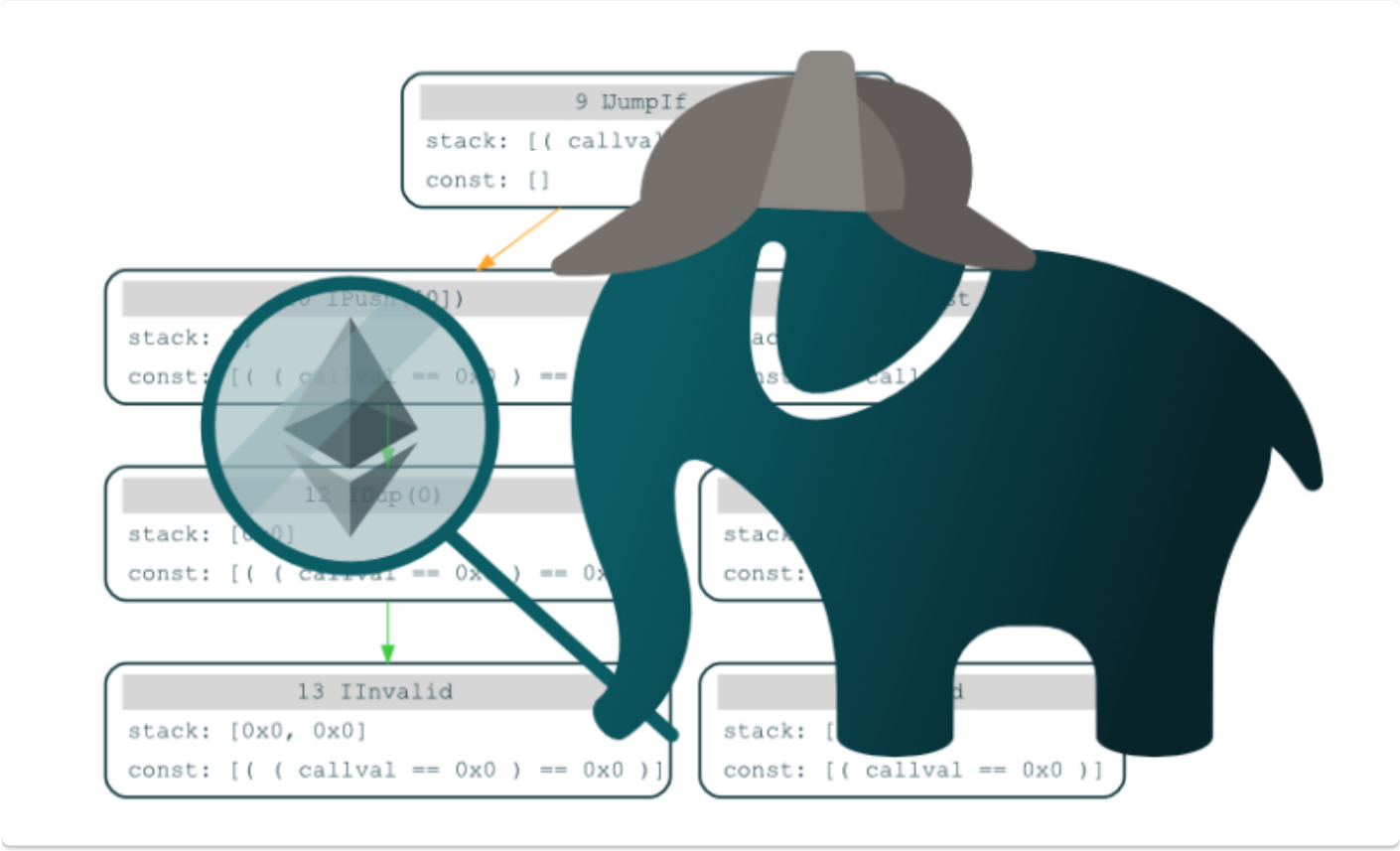
See Repo
See article

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

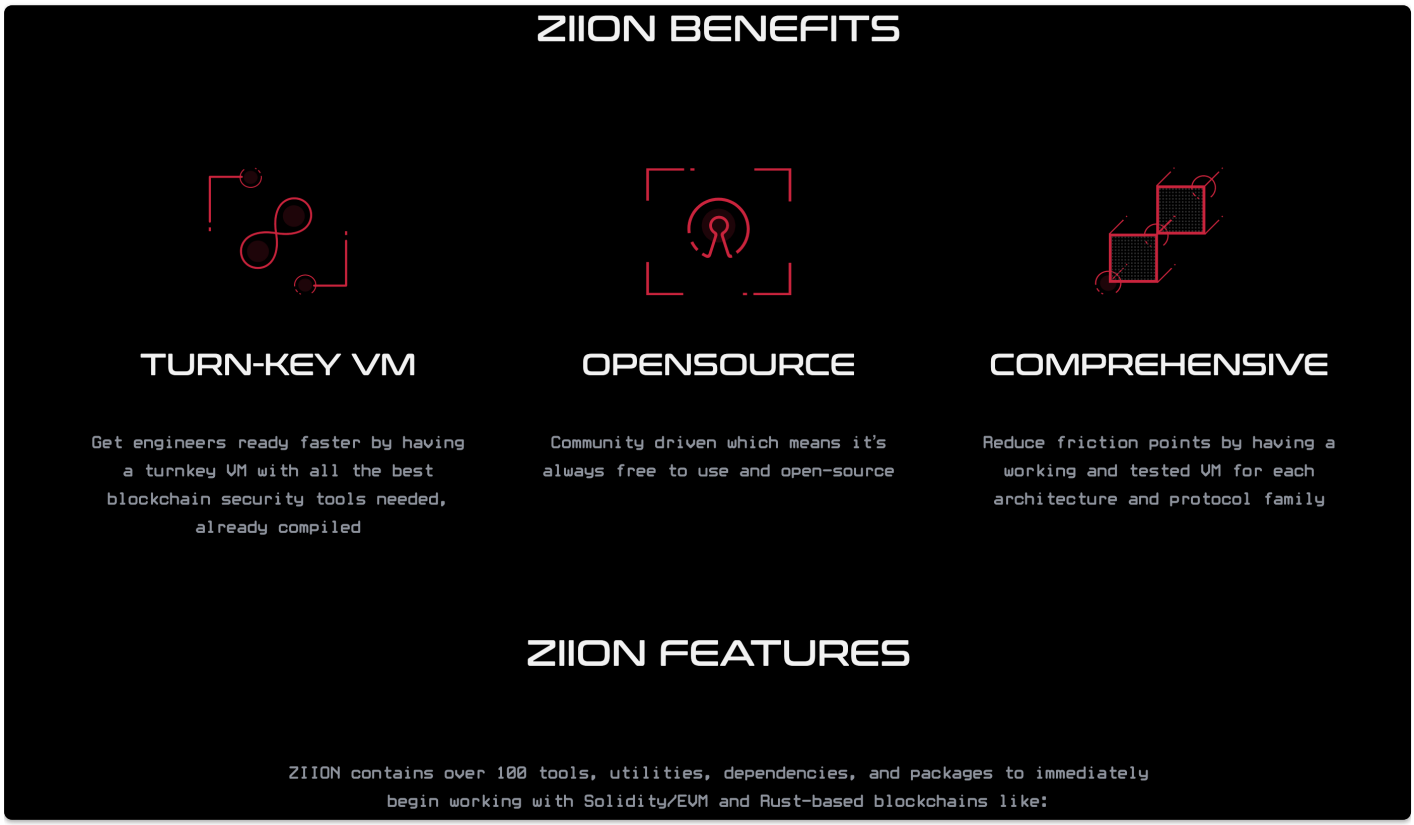
Install in docker with

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

ZIION from Halborn

See Docs

Download



Blockchains targeted

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

Tools include
Manticore, Slither, Mythril, 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:

```language-javascript
// Define the value(s) being fuzzed as an input argument
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:

```language-js
Running 4 tests for test/A.t.sol:ATest
[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` .

```language-js
// 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_-DXlXhOWCAoSpOx5PLocYgw

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 testCompatabilityOpenZeppelinProver(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.