

Lesson 23 - Formal Verification and Static Analysis

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.

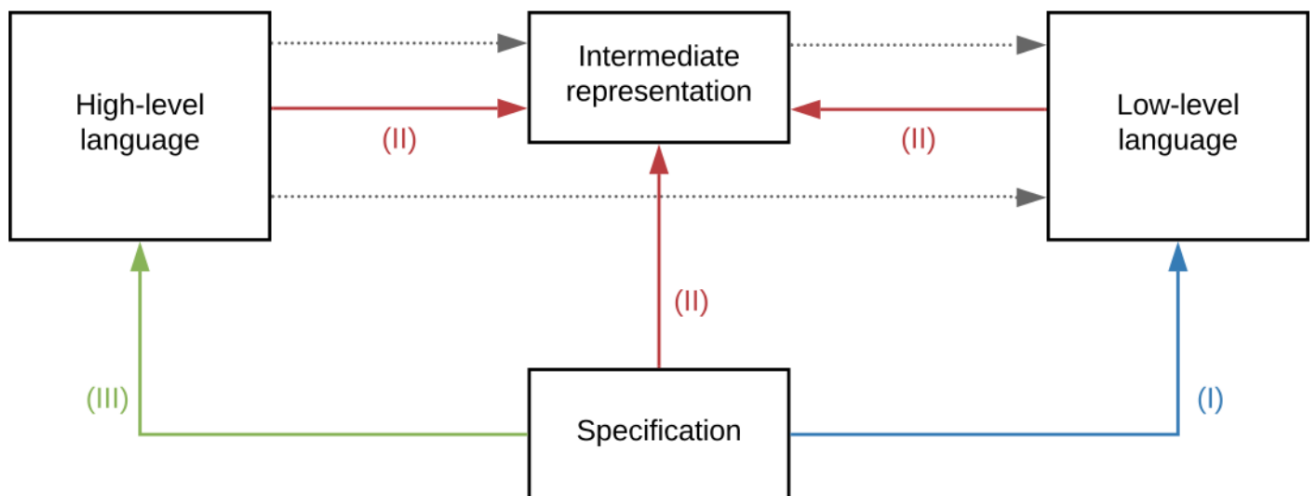
The formal verification methods are based on formal methods (mathematical methods), while the programming correctness methods are based on ensuring the programming as code is correct, which means the program runs without entering an infinite loop and gives correct outputs for correct inputs.

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 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.

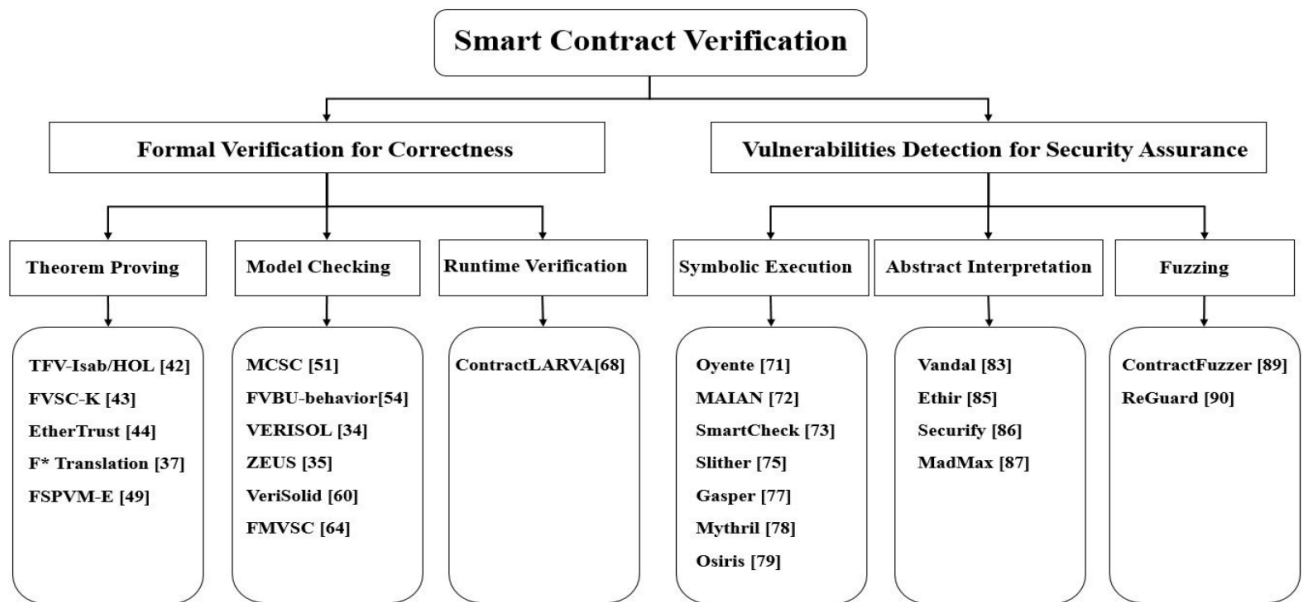
While no smart contract can be guaranteed as safe and free of bugs, a thorough code audit and formal verification process from a reputable security firm helps uncover critical, high severity bugs that otherwise could result in financial harm to users.

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 analyzer) scans the smart contract or its bytecode in order to understand its behavior and check for unexpected cases such as overflows and reentrancy vulnerabilities.

However, static analyzers are limited in the range of vulnerabilities they can detect. In a sense, a static analyzer 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.

Tools



Taxonomy of the smart contract verification's Tools. Source: [Verification of smart contracts: A survey](#)

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.

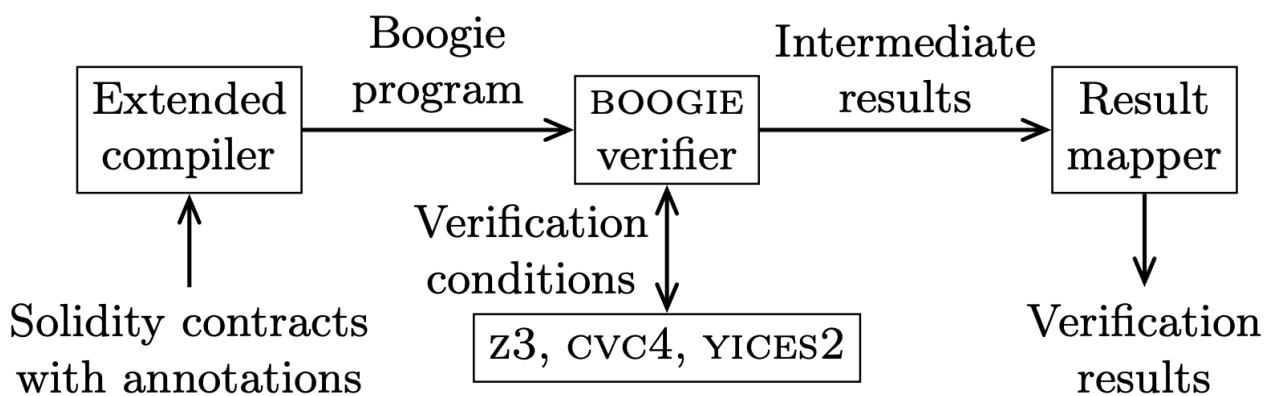
Solc-verify

[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.



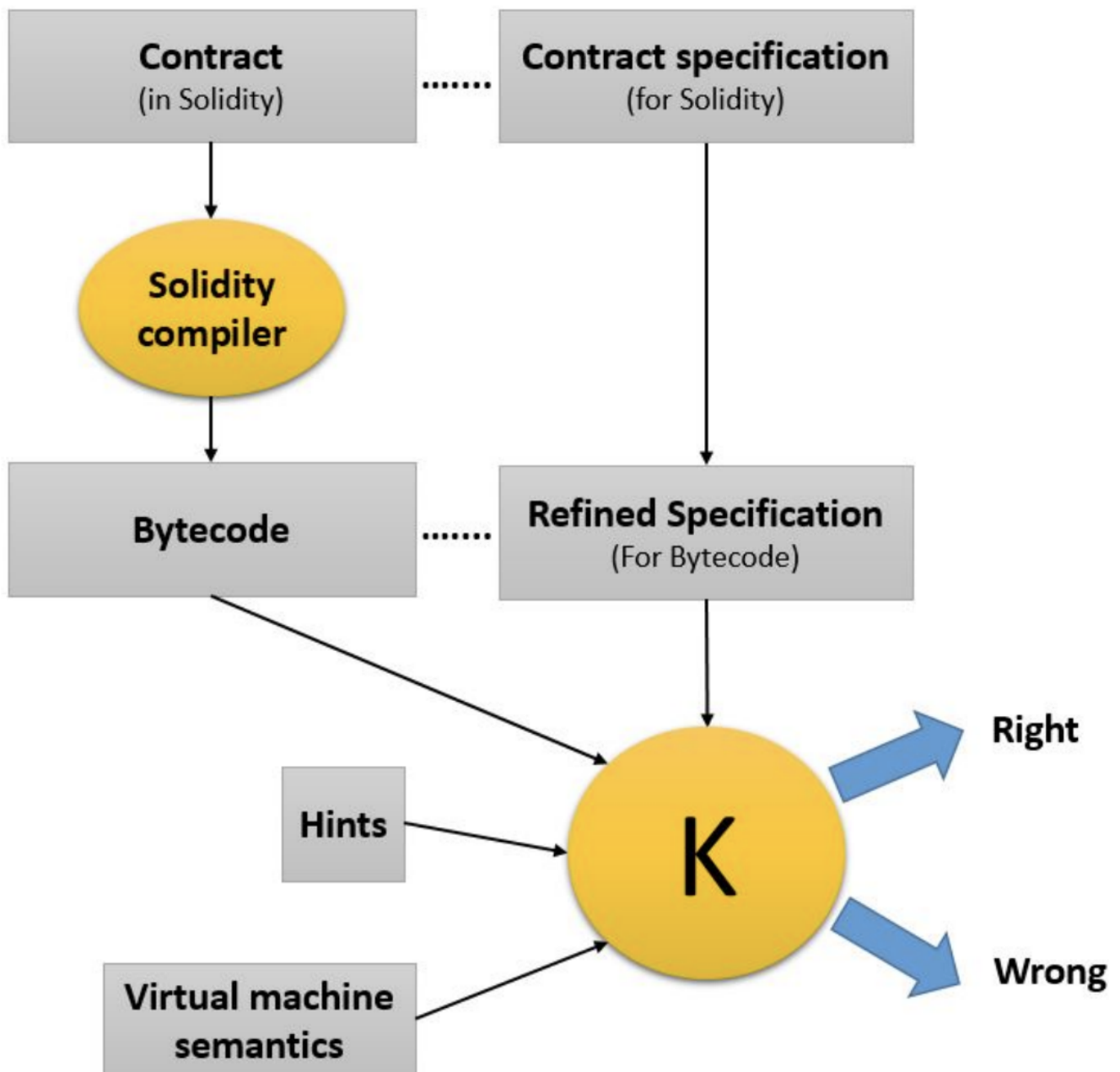
Overview of the solc-verify modules. The extended compiler creates a Boogie program from the Solidity contract, which is checked by the boogie verifier using SMT solvers. Finally, results are mapped back and presented at the Solidity code level.

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).

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
```

In Remix you can add

```
pragma experimental SMTChecker;  
to your contract, though this is deprecated.
```

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:

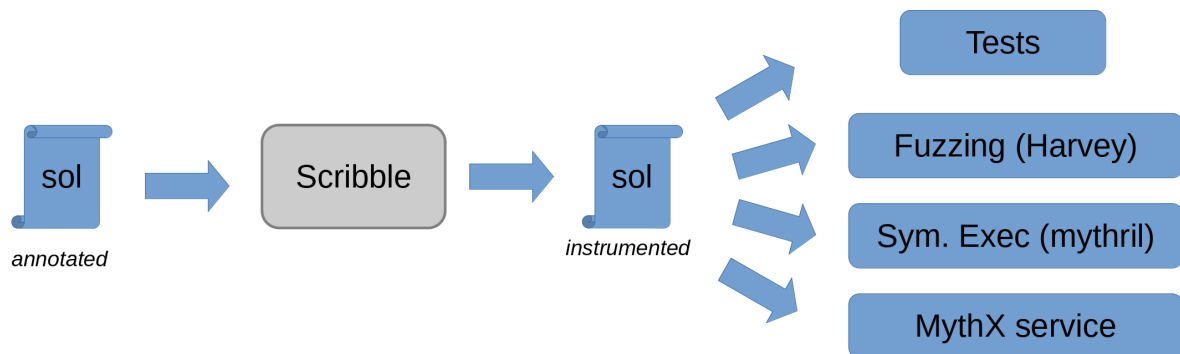
```
$ echidna-test myContract.sol
```

See further [examples](#)

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

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](#)
-



SLITHER

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

Example Detectors

Detectors

Num	Detector	What it Detects	Impact	Confidence
1	abiencoderv2-array	Storage abiencoderv2 array	High	High
2	arbitrary-send-erc20	transferFrom uses arbitrary from	High	High
3	array-by-reference	Modifying storage array by value	High	High
4	incorrect-shift	The order of parameters in a shift instruction is incorrect.	High	High
5	multiple-constructors	Multiple constructor schemes	High	High

Installation

Slither requires Python 3.8+ and [solc](#), the Solidity compiler.

Use pip3

```
pip3 install slither-analyzer
```

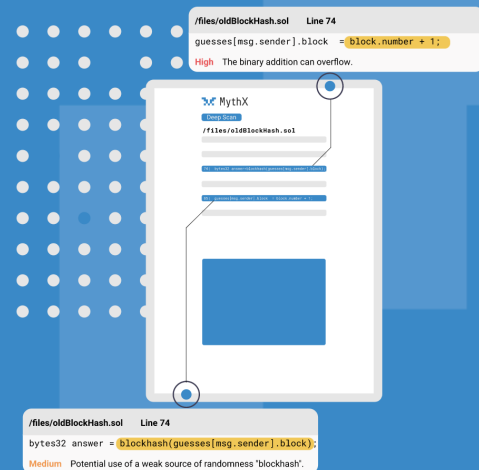
or git

```
git clone https://github.com/crytic/slither.git && cd slither
python3 setup.py install
```

Mythx

Smart contract security service for Ethereum

MythX™ by **ConsenSys Software Inc™** is the premier security analysis service for Ethereum smart contracts. Our mission is to ensure development teams avoid costly errors and make Ethereum a more secure and trustworthy platform.

[GET STARTED](#)


"MythX automatically scans for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. MythX's comprehensive range of analysis techniques—including static analysis, dynamic analysis, and symbolic execution—can accurately detect security vulnerabilities to provide an in-depth analysis report."

Based on SWC Registry

SWC-101

INTEGER OVERFLOW

SWC-106

UNPROTECTED SELFDESTRUCT

SWC-105

UNPROTECTED ETHER WITHDRAWAL

SWC-107

REENTRANCY

SWC-120

WEAK RANDOMNESS

SWC-110

ASSERT VIOLATION

SWC-124

WRITE TO ARBITRARY STORAGE LOC

SWC-127

JUMP TO ARBITRARY DESTINATION

SWC-109

UNINITIALIZED STORAGE POINTER

REPORT 4D5F97E8-EC0A-4757-BBB1-1C62BA475561
19:36 7/11/2019

[COPY FULL REPORT](#)

DETECTED ISSUES

2 High

4 Medium

2 Low

ID	SEVERITY	NAME	FILE	LOCATION
SWC-101	High	Integer Overflow and Underflow	old_blockhash.sol	L: 23 C: 37
SWC-105	High	Unprotected Ether Withdrawal	old_blockhash.sol	L: 33 C: 12
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	old_blockhash.sol	L: 29 C: 25
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	old_blockhash.sol	L: 23 C: 37
SWC-120	Medium	Weak Sources of Randomness from Chain Attributes	old_blockhash.sol	L: 27 C: 16
SWC-120	Medium	Weak Sources of Randomness from Chain	old_blockhash.sol	L: 32 C: 8

Mythx - CLI

See [Docs](#)

Installation

```
pip install mythx-cli
```

```
$ mythx analyze --help
```

```
Usage: mythx analyze [OPTIONS] [TARGET]...
```

Analyze the given directory or arguments with MythX.

Options:

<code>--async / --wait</code>	Submit the job and print the UUID, or wait for execution to finish
<code>--mode [quick standard deep]</code>	The MythX analysis mode to use
<code>--create-group</code>	Create a new group for the analysis
<code>--group-id TEXT</code>	The group ID to add the analysis to
<code>--group-name TEXT</code>	The group name to attach to the analysis
<code>--min-severity TEXT</code>	Ignore SWC IDs below the designated level
<code>--swc-blacklist TEXT</code>	A comma-separated list of SWC IDs to ignore
<code>--swc-whitelist TEXT</code>	A comma-separated list of SWC IDs to include
<code>--solc-version TEXT</code>	The solc version to use for compilation
<code>--solc-path PATH</code>	Path to a custom solc executable
<code>--include TEXT</code>	The contract name(s) to submit to MythX
<code>--remap-import TEXT</code>	Add a solc compilation import remapping
<code>--check-properties</code>	Enable property verification mode
<code>--scribble</code>	Enable scribble instrumentation (beta)
<code>--scribble-path PATH</code>	Path to a custom scribble executable (beta)
<code>--scenario [truffle solidity]</code>	Force an analysis scenario
<code>--help</code>	Show this message and exit.

Eth-Security-Toolbox

See [Repo](#)

It contains scripts to create a Docker container preinstalled and preconfigured with all of Trail of Bits' Ethereum security tools, including:

- [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

Available as a docker image

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

to share a directory

```
docker run -it -v /home/share:/share trailofbits/eth-security-toolbox
```

Open Zeppelin Defender

See [Docs](#)

Admin

Automate and secure all your smart contract administration.

Relay

Build with private and secure transaction infrastructure.

Sentinel

Monitor smart contracts and send notifications.

Autotask

Create automated scripts to call your smart contracts.

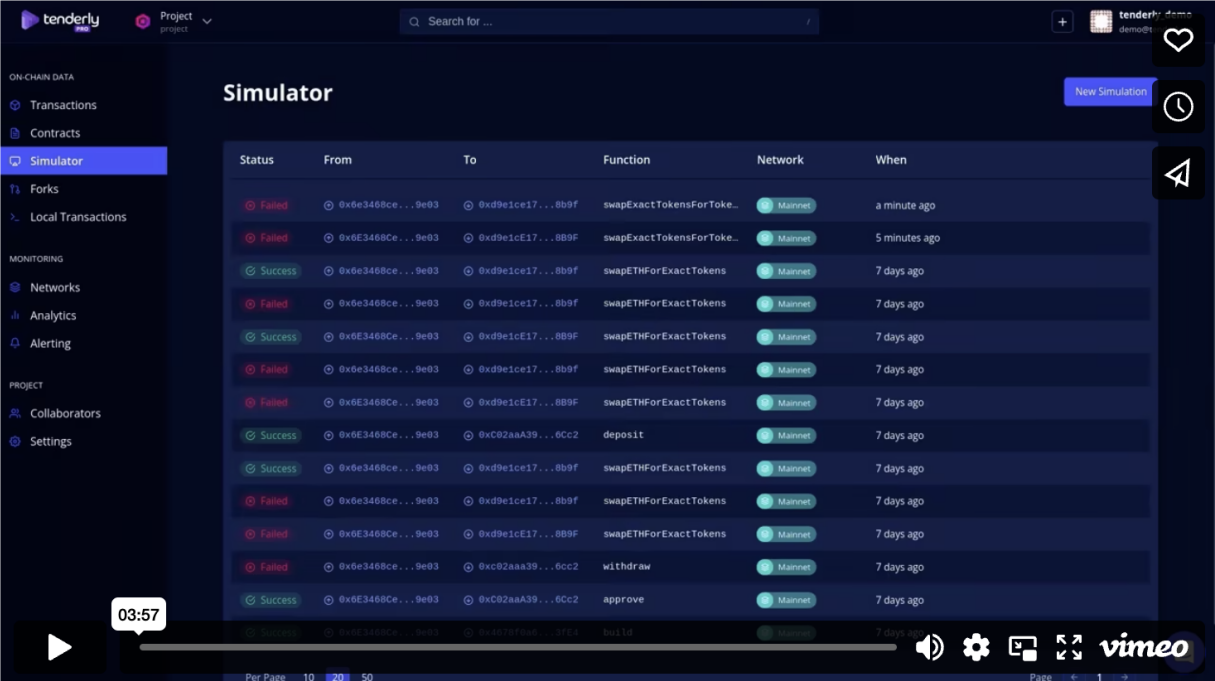
Advisor

Learn and implement security best practices.

Tenderly

![] (Screenshot%202022-11-23%20at%2018.15.07.png)

Simulator



The screenshot shows the Tenderly Simulator interface. On the left is a sidebar with navigation options: ON-CHAIN DATA (Transactions, Contracts, Simulator), MONITORING (Networks, Analytics, Alerting), and PROJECT (Collaborators, Settings). The main area is titled 'Simulator' and contains a table of transactions. A 'New Simulation' button is in the top right. A play button and a '03:57' timer are at the bottom left. The table has columns: Status, From, To, Function, Network, and When.

Status	From	To	Function	Network	When
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapExactTokensForToke...	Mainnet	a minute ago
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapExactTokensForToke...	Mainnet	5 minutes ago
Success	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Success	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Success	0x6e3488ce...9e03	0xc02aaa39...6cc2	deposit	Mainnet	7 days ago
Success	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Failed	0x6e3488ce...9e03	0xd9e1ce17...8b9f	swapETHForExactTokens	Mainnet	7 days ago
Failed	0x6e3488ce...9e03	0xc02aaa39...6cc2	withdraw	Mainnet	7 days ago
Success	0x6e3488ce...9e03	0xc02aaa39...6cc2	approve	Mainnet	7 days ago

If you ever used the JSON-RPC, you've probably stumbled upon [eth_call](#). If you haven't, it returns you the output of a Smart Contract function, without submitting an on-chain transaction. This can be useful in some cases, but it is quite limited.

With the Simulate tool, you can **provide the data for a transaction and see what would happen if you submitted the transaction right now**. So you get all of the tools we already offer for on-chain transactions **for the pending block**.

Not only that, but **you can even pick a specific block height, transaction index** and get the output of the transaction as if it happened at that specific point in time.

Sandbox for fast prototyping

sandbox

BETA

Straight O... ▾

Share ▾

Save ▾

Sign Up

Login

contract.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.14;
3
4 contract HelloWorld {
5     string public greeting;
6
7     constructor(string memory _greeting) {
8         greeting = _greeting;
9     }
10
11     function getGreeting() public view returns (string memory) {
12         return greeting;
13     }
14
15     function setGreeting(string memory _greeting) public {
16         greeting = _greeting;
17     }
18 }
```

<> script.js

```
1 // ethers is already injected
2
3 console.log("Fetching hello world contracts");
4 const contract = $contracts["HelloWorld"];
5
6 const provider = new ethers.providers.JsonRpcProvider($rpcUrl);
7 const factory = new ethers.ContractFactory(contract.abi, contract.evm.bytecode, prov
8
9 console.log("Deploying hello world contracts")
10 const helloWorld = await factory.deploy("Hello World")
11
12 console.log("Calling setGreeting function");
13 await helloWorld.setGreeting("Hello Sandboxes");
14
15 console.log(await helloWorld.getGreeting());
16
```

Network: Mainnet Block: Latest Compiler version: v0.8.14 Compiler Optimizations: 200

Configure RUN

Simulated Transactions Expand

You have to

▶ RUN

 Sandbox to see Simulated Transactions list

Console Output Expand

You have to

▶ RUN

 Sandbox to see Console Output