Jul 4, 2023 31 min read

An comprehensive overview of smart contract audit tools

Updated: Jul 10, 2023

Smart contract audit tools

Smart contract audit tools are used to identify security vulnerabilities in smart contracts. These tools can be used by smart contract auditors, developers or anyone else who wants to ensure that their smart contract is safe or has some level of security.

We will give an overview of static analyzers, mutation testing tools, fuzzing tools, formal verification tools, and visualization tools.
Note: some of these tools have incompatible Python requirements when installing, so take care when using them on the same machine. This article assumes proficiency with Solidity, so please see our tutorial learn Solidity if you are not yet familiar with the language.

Authorship

This article was written by Jesse Raymond (LinkedIn, Twitter) as part of the RareSkills Technical Writing Program.

1. Static analysis

Static analysis is a process of analyzing or scanning smart contract code to detect security vulnerabilities and ensure correctness without actually running the code.
Here are some popular static analyzers.

1a. Slither

Slither is a static analysis tool for smart contracts written in Python. It is one of the most popular static analysis tools for smart contracts, and is known for it's ability to find a wide range of vulnerabilities.
Slither detects vulnerable Solidity code, identifies where the error condition occurs in the source code of smart contracts and can analyze contracts written with Solidity >= 0.4.

Installation:
To install Slither, ensure you have python3.8+ and solc (Solidity compiler) installed and run the following commands:

```
pip3 install slither-analyzer
```

How to use:
You can use Slither in a Hardhat / Foundry / Dapp / Brownie project by running:

```
slither .
```

To target a single file (must not have imports) use the following command:

```
slither <path-to-contract>
```

Strengths:

- **Slither can find a wide range of vulnerabilities, including reentrancy attacks, timestamp dependency vulnerabilities, and integer overflow vulnerabilities.**
- **Slither is relatively easy to use.**
- **It is fast.**

Limitations:

- **Slither generates some false positives.**
- **Slither cannot find vulnerabilities that are caused by the interaction of different smart contracts(cross contract calls).**

1b. Consensys MythX

Consensys MythX is a cloud-based static analysis service for smart contracts. It can automatically scan for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. MythX can be used for static analysis, dynamic analysis, and symbolic execution and can detect security vulnerabilities to provide an in-depth analysis report.

Installation:
MythX is a cloud based service and requires a subscription to use. Head over to https://mythx.io/, create an account and subscribe to a desired plan to use the service.

How to use:
MythX has four official tools and libraries that it can be integrated with to use the service, which are the MythX CLI, Myth-JS, PythX and the MythX vscode extension.

Using MythX in Remix
MythX can also be used in Remix IDE, by enabling the MythX plugin on Remix and adding an API key.

- **Head to Remix and search for MythX in the plugin tab.**
- **Click activate to start using the plugin.**

- **Next, click on the MythX icon and click on the settings to add your MythX API key.**

- **You can now compile a smart contract of your choice and analyze it with MythX.**

Strengths:

- MythX is easy to use.
- MythX offers a comprehensive report.
- MythX can be integrated with other tools

Limitations:

- It is a commercial tool and requires subscription.
- MythX cannot find vulnerability that involves interactions with other smart contracts (cross contract calls).
- It is cloud based and not all auditors or developers like using cloud based services.

1c. Rattle

Rattle is a static binary EVM analysis framework that produces a control flow graph after removing the DUP, SWAP, PUSH , and POP opcodes. This reduces the number of EVM instructions and thus makes the graph much more readable.
Rattle takes EVM bytecode and recovers the original control flow graph of the contract so users can be more farmiliar with contracts that they interact with.

Installation:

Required dependencies:

- python3
- graphviz
- cbor2
- pyevmasm

To proceed with installation,

- Check python version (no version, install Python3)

```
python -V
#or
```

```
python3 -V
```

- **Install graphviz:**

Linux:

```
sudo apt-get install graphviz
```

macOS:

```
brew install graphviz
```

- **Clone Rattle repository and enter project:**

```
git clone https://github.com/crytic/rattle.git
cd rattle
```

- **Run the following commands to install the dependencies in a in Python virtual environment(Make sure to run this command everytime before using rattle):**

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

(The above command will create a Python virtual environment which is an isolated environment for Python used for running and installing Python packages without affecting globally installed packages, this is why it is important to run the command each time we use Rattle).
How to use:
Once inside the rattle/ folder, run the following command to test Rattle on the KingOfTheEtherThrone contract bytecode.

```
python3 rattle-cli.py --input
inputs/kingofether/KingOfTheEtherThrone.bin -O
```

You can also generate bytecode for any smart contract with:

```
 solc --bin-runtime <path-to-file> 2>/dev/null | tail -n1 > <desired-
destination-path>/contract.bin
```

And analyze it with rattle using:

```
#load up dependencies first
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt

#run rattle
python3 rattle-cli.py --input <desired-destination-path>/contract.bin
```

(Note that this must be inside the rattle repository folder)

As an example, if we analyze the bytecode of the Storage contract in Remix default environment, we can see the control flow of the “store(uint256)” function below:

We can see that the flow of the "store(uint256)" function, how many storage slot it reads and write from (slot 0 in this case) and occasions where a revert can occur (e.g. having less calldata than required).

Strengths

- **Rattle is useful to visualize the control flow of smart contracts and trace function execution in the smart contract.**
- **It can trace both the stack and memory of smart contracts.**
- **It gives a simple function call analysis, e.g. it can identify and notity if a contract can send ether from function or not.**
- **It uses Single Static Assignment form to make control flow graph interface friendlier.**

Limitations

- **Since its mainly used to analyze the flow of smart contracts, it is not perfect, and it can miss some bugs and vulnerabilities as it only analyzes smart contract bytecode and can miss vulnerabity that may be related to business logic, transaction order dependence, etc.**
- **It is not always easy to understand the results of the analysis.**

(Note: rattle does not work with Solidity 0.8.20).

1d. Mythril

Mythril is both a static and dynamic analysis tool for Solidity smart contracts. It is designed to detect potential vulnerabilities and security weaknesses in Solidity code and can detect vulnerabilites that involves contract interactions like reentrancy. It supports smart contracts built for Ethereum, Hedera, Quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains.

Installation:

To install Mythril, run the following commands:

Linux:

```
# update packages
sudo apt update

# Install solc
sudo apt install software-properties-common
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt install solc

# Install libssl-dev, python3-dev, and python3-pip
sudo apt install libssl-dev python3-dev python3-pip

# Install mythril
pip3 install mythril

# Check version of Mythril
myth version
```

macOS:

```
# Update Homebrew and packages
brew update
brew upgrade

# Install solc
brew tap ethereum/ethereum
brew install solidity

# Install mythril
pip3 install mythril

# Check version of Mythril
myth version
```

How to use:
To test a Solidity smart contract with Mythril run:

```
myth analyze <path-to-file>
```

Testing Mythril
To visualize how Mythril works, we have created Solidity with code that is vulnerable to reentrancy. We will test this code with Mythril to see if it picks up the vulnerabilty.
Paste the code below in a Solidity file:

```
// SPDX-License-Identifier: GPL-3.0-or-laterpragma solidity
^0.8.0;contract Vulnerable {
    mapping(address => uint) public balance;

    function donate(address to) public payable {
        balance[to] += msg.value;
    }

    function withdraw(uint amount) public {
        if (balance[msg.sender] >= amount) {
            msg.sender.call{value: amount}("");
            balance[msg.sender] -= amount;
        }
    }

    function queryBalance(address to) public view returns (uint) {
        return balance[to];
```

```
        }
    }
```

When we run myth analyze <path-to-file>, we get the following report:
(open in a new tab and zoom in to see details).

Mythril was able to find five vulnerabilties in the smart contract.
Mythril usually runs three transactions when carrying out analysis.
To get better results we can increase the number of transactions with the -t <tx count>.

For example, to run four transactions while testing the Solidity file above, we can run:

```
myth analyze <path-to-file> -t 4
```

This number parameter can be increased for better results. However it will slow down the test time.
Read more about it here.

Strengths

- **Mythril can find a wide range of vulnerabilities including: reentrancy attacks, integer underflow/overflow and some more vulnerabilities under the Smart Contract Vulnerability Classification Registry.**
- **It is easy to use.**
- **It is a static/dynamic analysis tool, hence it can find some vulnerabilties that involves interactions with smart contracts.**

Limitations

- **Mythril is slow, and the higher the test transaction count, the slower the test.**
- **It can give false positives.**

1e. Securify v2.0

Securify 2.0 is a security scanner for Ethereum smart contracts. It is the successor of the Securify (deprecated) security scanner.

Installation:

The following dependencies must be installed to use Securify 2.0 (Note that securify works with just Python 3.7 and does not require the previous securify to installed):

- **Solc**
- **Souffle**
- **Graphviz / Dot**

To install the dependencies, follow the instructions below:

Linux:

```
# Install solc
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc

# Install graphviz
sudo apt install graphviz
```

macOS:

```
# Install solc
brew tap ethereum/ethereum
brew install solidity

# Install graphviz
brew install graphviz
```

Install Souffle from the repository or see build instructions here

After downloading all the dependencies, run the following commands to clone securify 2.0 repository and install in a virtual environment:

```
git clone https://github.com/eth-sri/securify2.git

cd securify2/securify/staticanalysis/libfunctors

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:`pwd`

cd ../../../

python3 -m venv venv # Run this only if `source venv/bin/activate`
fails to run
```

```
virtualenv --python=/usr/bin/python3.7 venv

source venv/bin/activate

# Run securify 2.0
securify -h
```

How to use:
To analyze a smart contract with securify 2.0, run:

```
securify <path-to-file> [--use-patterns Pattern1 Pattern2 ...] # Run
securify -l to see all available patterns
```

Strengths

- **Securify v2.0 can find a wide range of vulnerabilities, including state shadowing, uninitialized storage, locked ether, unrestricted delegatecall and reentrancy attacks.**
- **It can find up to 37 different smart contract vulnerabilities.**

Limitations

- **It is not actively maintained.**
- **Supports only flat contracts (contracts with no imports).**
- **Uses old version of python**
- **Installation process is tedious.**
- **It can produce false positives.**

2. Mutation Testing

Mutation testing evaluates and improves test suites by introducing faults (mutants) into the program being tested.

The goal is to see how well the test suite detects and eliminates these mutants. Live mutants, which pass all tests, expose weaknesses in the test suite and guide its improvement.

You can learn more about mutation testing from our blog post on mutation testing.
Let us look at some mutation tools.

2a. Vertigo-rs

Vertigo-rs by RareSkills is an actively maintained fork of Joran Honig's vertigo testing framework with added support for Foundry.
Vertigo-rs will mutate files in src/ and run forge test to see if the mutant survives. Files that end with .t.sol or have test in their name (including the path) are ignored. Files in lib (or any directory that isn't src/) will not be mutated.
You do not need to specify that you are in a Foundry project. The presence of a foundry.toml file will signify to this tool that you are in a Foundry repo. If you have configuration files for Truffle or Hardhat in your project, this tool create an error.

Installation:
To install vertigo-rs, follow the following steps:

```
git clone https://github.com/RareSkills/vertigo-rs
cd vertigo-rs
python setup.py develop
```

How to use:
Use these commands to run vertigo-rs on a Foundry project:

```
cd <your foundry project>

# Run vertigo
python <path-to-the-vertigo-project>/vertigo-rs/vertigo.py run
```

We'll try out an example below.
Initialize a new Foundry project with:

```
forge init vertigo-test
cd vertigo-test
```

We will override the default src and test files in the project so that code mutants can be easily generated

Copy and paste this code inside the Counter.sol file.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
contract Counter {
    int256 public number;

    function setNumber(int256 newNumber) public {
        number = newNumber;
    }

    function increment() public {
        number = number + 1;
    }

    function decrement() public {
        number = number - 1;
    }
```

```
    function multiply() public {
        number = number * 2;
    }
}
```

Paste this inside the Counter.t.sol file.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "../src/Counter.sol";

contract CounterTest is Test {
    Counter public counter;

    function setUp() public {
        counter = new Counter();
        counter.setNumber(0);
    }

    function testIncrement() public {
        counter.increment();
        counter.increment();
        assertEq(counter.number(), 2);
    }

    function testDecrement() public {
        counter.setNumber(20);

        counter.decrement();
        counter.decrement();
        assertEq(counter.number(), 18);
    }

    function testMultiply() public {
        counter.setNumber(6);
        counter.multiply();

        assertEq(counter.number(), 12);
    }
```

```
        function testSetNumber(int256 x) public {
            counter.setNumber(x);
            assertEq(counter.number(), x);
        }
    }
```

Next, run python3 <path-to-vertigo-rd-folder>/vertigo-rs/vertigo.py run. We get the following result:

We can see that three mutants were generated and none survived because our test was able to pick them.

Let us test for a scenario where a mutant survives. If mutating the code does not cause a test to fail, this means that the test suite has been poorly written.
Comment out the assertEq(counter.number(), 2); on line 18 of the test file, inside the "testIncrement" function and run vertigo again.
This time we got a surviving mutant:

2b. Certora Gambit

Gambit is a sophisticated Solidity mutation system. It utilizes mutation operators to modify the source code of a Solidity program. This creates mutants, which are mutated variants of the program.

These mutants are useful for analyzing test suites and specifications used for formal verification. Each mutant represents a potential bug. The efficiency of test suites and specifications can be

measured by the number of remaining mutants detected.

Installation:
Gambit requires both Rust and solc to be installed.
Run the following commands:
Install Rust & cargo:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Install solc:
linux:

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

macOS:

```
brew tap ethereum/ethereum
brew install solidity
```

Now to install Gambit, clone the Gambit repository and build Gambit:

```
git clone https://github.com/Certora/gambit.git
cd gambit/
cargo install --path .

gambit -h
```

How to use:
To create mutants, run:

```
gambit mutate --filename <file.sol>
```

This will generate mutants for the Solidity file in a gambit_out/mutants/ folder.
You can get a summary of the mutants generate by running:

```
gambit summary
```

As an example, we will generate mutants for the Solidity code below (name it Code.sol).

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.0;
contract Vulnerable {
    mapping(address => uint) public balance;
```

```
    function donate(address to) public payable {
        balance[to] += msg.value;
    }

    function withdraw(uint amount) public {
        if (balance[msg.sender] >= amount) {
            msg.sender.call{value: amount}("");
            balance[msg.sender] -= amount;
        }
    }

    function queryBalance(address to) public view returns (uint) {
        return balance[to];
    }
}
```

When we run gambit mutate --filename Code.sol, we get:

Seven mutants were generated and we can find them in the gambit_out/ folder as stated earlier.

Now run, gambit summary. We get the summary of the mutants generated:

Strengths

- **Easy installation process and usage.**
- **Gambit also generates comments in mutant files that indicates which part of the code was changed.**
- **Mutants are generated fast.**
- **You can mutate multiple Solidity files using configuration files.**

Limitations

- **Unlike vertigo-rs that mutates smart contract code and run the test for it, Gambit only mutates the Solidity code and you would have to manually run test on the code to see if any mutants survives.**

2c. SuMo

SuMo is a mutation testing tool for Solidity Smart Contracts, designed to run mostly in the NodeJS environment.

Installation:

SuMo can be used in a Truffle, Hardhat, Brownie or Foundry project, but we will be using hardhat.

First we create a hardhat project and install sumo as an npm package inside our project:

```
mkdir sumo-hardhat
cd sumo-hardhat
npm init -y
npm install --save-dev hardhat
npx hardhat init
npm install @morenabarboni/sumo
```

A configuration file will be generated: sumo-config.js.
Replace the auto generated config with the following:

```
module.exports = {buildDir: "artifacts",contractsDir:
"contracts",testDir: "test",skipContracts: [""],skipTests:
[""],testingTimeOutInSec: 300,network: "none",testingFramework:
"hardhat",minimal: false,tce: false,};
```

SuMo is now ready to be used.

How to use:
Now that our configuration is done, we can run:

- **npx sumo preflight to view the available mutations and save a sample report to .sumo/report.txt.**
- **npx sumo mutate to create mutants. Mutants are saved to .sumo/mutants and a sample report will be available in .sumo/report.txt,**

Run npx sumo list to view all enabled mutation operators, and npx sumo enable to enable all mutation operators.

The .sumo folder is created for the output of these commands. Click here to learn more about the tool.

Hardhat comes with a default contract and test for. We can test the reliability of the test by using sumo to find mutants.

To do this we run:

```
npx sumo test
```

After the test, we get the following report:

We received a mutation score of 72.92%. Although this is the default file that comes with hardhat, it is better to have a mutation score of at least 80%.

You can check the .sumo/ folder for more information as well as the original contract and test can be found in the.sumo.baseline folder.

Strengths

- **Supports Truffle, Hardhat, Brownie and Foundry projects.**
- **It features 25 Solidity-specific mutation operators, as well as 19 traditional operators.**
- **SuMo generates many mutated codes.**
- **Has unique features, such as preflight, mutate, restore and diff.**
- **SuMo generates helpful reports after running commands like preflight and mutate.**
- **It generates a mutation score and test summary at the end of testing.**

Limitations

- **As of writing, SuMo is specifically designed to operate as a development dependency within an npm project and does not support global usage.**
- **It may be slow in large test.**

3. Fuzzing and Invariant Testing

Fuzzing or fuzz testing is another technique that can be used for finding vulnerabilities in smart contracts. In a fuzz test, the functions of a smart contract are called with unexpected and random inputs. This can expose scenarios in which the system breaks.

Invariant testing, on the other hand focuses on verifying specific properties or invariants. These are conditions that must always be true throughout the use of a smart contract.

Both test methods are dynamic analysis, as the smart contract code is actually executed during the test.

3a. Consensys Diligence Fuzzing

ConsenSys Diligence Fuzzing is a fuzz testing tool for identifying flaws in smart contracts. It is part of the ConsenSys Diligence security tool suite.

ConsenSys Diligence fuzzing has a CLI tool that allows users to run fuzzing locally.
It also has a cloud-based dashboard, which provides a centralized platform for organizing and monitoring fuzzing activities. This includes valuable tools for assessing the results of a fuzzing

campaign.

Installation:
The fuzzing CLI runs on Python 3.6+, including 3.8 and pypy3.
To install the Diligence fuzzing CLI run:

```
pip3 install diligence-fuzzing
```

Visit https://fuzzing.diligence.tools/login to create an account and a fuzzing campaign project (Cloud based testing).

How to use:
Diligence fuzzing leverages smart contract annotations written in Scribble, which is a language designed for specifying security properties and constraints.
The fuzzer checks these property and reports any violations.
To learn more about scribble, visit the documentation.

Scribble annotations are usually in this particular structure:

```
if_succeeds {:msg "<Description of property>"} <boolean scribble
expression>;
```

- The if_succeeds is the start of the annotation and indicates that a certain property should hold after a function is executed successfully.
- {:msg "<Description of property>"} is an optional part where we can describe the property they are specifying. This helps creates better for better documentation and understanding of the annotation.
- <boolean scribble expression> is the core part of the annotation which represents the property or constraints we want to specify. They can be x >= y, balanceOf(msg.sender) <= totalSupply, etc.

For a correctly implemented ERC20 token transfer, a simple scribble annotation would look like this:

```
contract ERC20example {
    /// if_succeeds {:msg "ERC20 transfer"} balance(msg.sender) ==
old(balanceOf(msg.sender)) - amount;
    function transfer(address recipient, uint256 amount) public {
        // ...
    }
}
```

Here we are saying if the transfer succeeds, the balance of the sender should have been deducted balance(msg.sender) == old(balanceOf(msg.sender)) - amount.

The fuzzer will try to find random inputs that would break this property.
You can learn more about creating fuzz campaings and writing annotations here.

Using Diligence fuzzing with foundry
Diligience fuzzing also supports foundry fuzz test. We can create a foundry fuzz test project and use the Diligence fuzz CLI to create a campaign for our foundry test.
Let us see an example below.

Run the following command to create a new foundry project:

```
forge init diligence-fuzz-foundry
cd diligence-fuzz-foundry
```

Once we have our project created, update the sample code as follows:
Rename Counter.sol in the src folder to Password.sol and paste this:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
contract Password {
    // Secret password
    uint256 private password;

    constructor() payable {
        require(msg.value >= 1 ether);
        // Set secret password as 25
        password = 25;
    }

    function withdraw(uint256 password_) public {
        // If anyone guesses the correct password, they can take all
the ether in this contractif (password_ == password) {
            (bool success, ) = msg.sender.call{value:
address(this).balance}(
                ""
            );
            require(success, "failed to send");
        }
    }

    function balance() public view returns (uint) {
        return address(this).balance;
    }
```

```
        receive() external payable {}
    }
```

Rename Counter.t.sol in the test folder to Password.t.sol and paste this:

```
    // SPDX-License-Identifier: UNLICENSEDpragma solidity ^0.8.13;import
    "forge-std/Test.sol";
    import "../src/Password.sol";

    contract PasswordTest is Test {
        Password code;

        function setUp() public {
            // Deploy the contract with 1 ether
            code = new Password{value: 1 ether}();
        }

        function testFuzz_Password(uint256 password_) public {
            // We are fuzzing the `password_` input,// foundry fuzzer will
    try to find a number that matches our password and breaks the assert
    statement below,// which means all the ether from the contract has been
    taken
            code.withdraw(password_);
            assert(code.balance() == 1 ether);
        }

        receive() external payable {}
    }
```

We won't be testing this with foundry as it is not the aim of this section.
We have everything we need set up. Next, head over to the Diligence fuzzer dashboard and create an API key.

Click on “Create new API key”

Copy the API key and run this command in the foundry project:

```
fuzz forge test -k <your-api-key-here>

# Without the `<>` sign
```

We get this result when we run the command.

We can click on the link printed on the terminal, this will take us to the campaign, as shown below:

We have a failed property!

The fuzz campaign was able to detect the vulnerability in the “testFuzz_password” function as shown in the image above.

To get more details about the failed property, we can click the expand button.

We see the underlined assert(code.balance() == 1 ether);, where the fuzzer was able to break this property by fuzzing the secret password.

We can also see that “25” is passed to the “testFuzz_Password” in the transaction steps below

To learn more about using diligence fuzz with foundry, visit the documentation here.

Strengths

- Diligence fuzzer is effective in finding bugs.
- The diligence fuzz CLI comes with help commands to for fuzz testing, such as fuzz arm to instrument scribble annotated Solidity contracts and disarm to revert the changes back to the un-instrumented state.
- It is scalable and can be used to fuzz large codebases.
- Fuzzing can be done locally or in the cloud.
- Has support for foundry fuzz test.
- The cloud based service is automated and manages the infrastructure.

Limitations

- It does not have an easy set up.
- Testing can be slow at times.
- It requires a subscription and is not totally free to use.
- Part of this tool is cloud-based and centralized, so there is a concern about data privacy.
- It is still in development and does not have an active community support.

3b. Foundry Invariant Testing

Foundry is a fast smart contract development tool built with Rust.
It handles project dependencies, compiles projects, runs tests, deploys, and allows you to interact with the blockchain via the Solidity scripts and command line.
Installation:
Install foundry:

```
curl -L https://foundry.paradigm.xyz | bash
```

Restart the terminal and run foundryup to install Forge, Anvil, Cast, and Chisel.
If you run into any problems, check out the FAQs.

How to use:
We have a comprehensive tutorial on this, learn about invariant testing with foundry here.

Strengths

- **Testing with Foundry is fast.**
- **There are helpful cheat codes and customizations that can be used to write more efficient tests, such as increasing the number of times a fuzz test runs.**
- **Foundry also has a handler-based testing method which is helpful for performing an invariant test that involves cross-contract interactions.**

Limitations

- **Manual bounding of input range: Sometimes you may need to manually bound the range of fuzzed input random numbers in order to achieve a desired result. This is because Foundry may not be able to pick up the right input value on its own.**

3c. Echidna

Echidna is a fuzzing and property-based security testing tool for Ethereum smart contracts written in Haskell.

It falsifies user-defined predicates or Solidity assert statements using sophisticated grammar-based fuzzing campaigns based on a contract ABI.

Installation:
Echidna requires Slither to be installed, so ensure to have Slither installed before moving on with the installation.

If Slither has not previously been installed, run the following command:

```
pip3 install slither-analyzer
```

To install Echina, run:

```
brew install echidna
```

Or build from binaries.

How to use:
Echidna is a reliable fuzzing tool for smart contracts. It analyzes smart contracts based on predefined invariants.

It generates random sequences of contract calls to test against the invariants. If Echidna discovers a call sequence that violates an invariant, it reports it. If no such sequence is detected, it gives confidence that the contract is safe based on the aspects of the predefined invariants Although this does not mean that there will be no defects in the code.

Echidna invariants are specified by a function.

This functions starts with the echidna_ keyword, it doesn't take any input parameters and must return a boolean. Through this, Echidna knows what invariant it will try to break.

Let's see an example below:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Code {
    address public owner = address(0x11EF);

    modifier onlyOwner() {
        require(msg.sender == owner, "not owner");
        _;
    }

    function changeOwner(address newOwner) public {
        owner = newOwner;
    }

    // Invariantf
    unction echidna_no_address_zero() public returns (bool) {
        return owner != address(0x00);
    }
}
```

When we run Echidna, it will try to falsify the invariant inside the "echidna_no_address_zero" function.

Run this command:

```
echidna Code.sol
```

We get this output:

We defined an "onlyOwner" modifier and forgot to use it in the "changeOwner" function.

Therefore, anyone could call the function and set address(0x00) as the owner, thus breaking the invariant owner != address(0x00).

Echidna was able to spot this bug and changed the owner address.

See this repo if you with to use Echidna with foundry.

Strengths

- **Echidna is a powerful fuzzer that can find vulnerabilities in smart contracts.**

- Echidna can test contracts compiled with different smart contract build systems, including Truffle or Hardhat using crytic-compile. To invoke echidna with the current compilation framework, use echidna .
- Echidna is easy to use and can be run from the command line.

Limitations

- Echidna can be slow in detecting flaws in large contracts.
- It has limited support for testing contracts that rely heavily on external libraries, which may limit its testing capabilities in such cases.
- It's support for the Vyper programming language is limited.
- Echidna is not perfect and could miss some bugs in smart contracts.

3d. Dapptools

Dapptools is a suite of Ethereum focused CLI tools. It includes a static analysis tool for smart contracts, as well as a number of other tools for testing and debugging smart contracts such as:

- dapp - build, test, fuzz, formally verify, debug & deploy solidity contracts.
- seth - Ethereum CLI. Query contracts, send transactions, follow logs, slice & dice data.
- hevm - Testing oriented EVM implementation. Debug, fuzz, or symbolically execute code against local or mainnet state.
- ethsign - Sign Ethereum transactions from a local keystore or hardware wallet.

Installation:
Install Nix first:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Now use Nix to install Dapptools:

```
nix profile install github:dapphub/dapptools#{dapp,ethsign,hevm,seth}
```

If you run into issues while trying to install Dapptools on a mac, here's a great step to follow https://roycewells.io/writing/dapptools-m1/.

How to use:
You can use Dapptools for unit testing, fuzz or property based testing, invariant testing and even symbolic execution.
We will be doing a fuzz and invariant test.

Fuzz test
Run the following commands:

```
mkdir dapp-fuzz-test
cd dapp-fuzz-test
dapp init
```

Now delete the default Solidity files inside the src/ directory.
Create two new files, Code.sol and Code.t.sol and paste the following code respectively:

- Code.sol

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.0;
contract Code {
    uint256 private password;

    constructor() payable {
        require(msg.value >= 1 ether);
        password = 25;
    }

    function withdraw(uint256 password_) public {
        if (password_ == password) {
            (bool success, ) = msg.sender.call{value:
address(this).balance}(
                ""
            );
            require(success, "failed to send");
        }
    }

    function balance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Code.t.sol

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.6;
import "ds-test/test.sol";

import "./Code.sol";

contract CodeTest is DSTest {
    Code code;

    function setUp() public {
        code = new Code{value: 1 ether}();
    }
```

```
        function test_fuzz(uint256 password_) public {
            code.withdraw(password_);
            assert(code.balance() == 1 ether);
        }

        receive() external payable {}
    }
```

We are deploying the Code contract with 1 ether and 25 as a secret number.
To withdraw this Ether, the fuzzer must pass in the correct secret number which is a private variable.
Note that the fuzzer does not read storage slots and just passes random numbers to break the assert(code.balance() == 1 ether); inside the test_fuzz(uint256 password_) test function.

Fuzz test functions in Dapptools usually takes in at least one parameter. This parameter will be random on each run of the test.
Run the test with dapp test.

Dapp automatically knows that it is a fuzz test and gives this result:

We can see that the test passed and the fuzzer was not able to get the correct secret number.

The go to way is to increase the number of runs for the fuzzer.
To do this we will use the --fuzz-runs <number> flag.
For example if we rerun the test with, dapp test --fuzz-runs 3000, we get the following:

We an increased number of runs, the fuzzer was able to get our secret number.
We can replay the transaction with dapp test --replay '("test_fuzz(uint256)","0x0000000000000000000000000000000000000000000000000000000000000019")' --verbosity 3.
The --verbosity <number> flag is to increase the test verbosity so we can see the transaction traces.

We see that the Code.balance() function returned a zero, which caused the test to revert and fail.
Invariant testing in dapp also works similar to that of Foundry, learn more here.

Strengths

- With dapp, we can perform unit test, fuzz test, invariant test, symbolic execution and test against and rpc state.
- We can replay failed test and even debug it to see the execution on the EVM, visualize the stack and memory.
- Dapp does not execute transactions using rpc. Instead, it directly calls the hevm cli. This is faster and provides a lot of versatility that rpc does not, such as fuzz testing.
- It supports cheatcodes to make testing easier, like hevm.warp(uint256), and hevm.load(address,bytes32).

Limitations

- It is complex to use and debug information can be difficult to understand.
- It has limited community support.
- dapp can be slow to run.
- It has poor documentation.
- dapp fuzzer is slow to find common vulnerabilities in small runs.

Formal Verification and Symbolic Execution

Formal verification is the process of proving and disproving the correctness of a program (smart contracts in this case) using mathematical proofs with respect to a formal specification.
Formal verification is a broad term that encompasses different techniques. Three of the most common techniques are **model checking**, **theorem proving**, and **symbolic execution**.

- Model checking creates a model of the system (smart contracts in this case) by constructing a state transition diagram or a finte state machine and then the model checker checks the model to see if it satisfies a set of desired properties.
- Theorem proving uses mathematical proofs to show that the system satisfies a set of desired properties.
- Symbolic execution represents the system as a set of symbolic expressions (using symbols like x and y, instead of concrete values like 5 and 6) and then executes the system on these expressions to see if it reaches any states that violate the desired properties.

Some of the tools here use a combination of one or more formal verification techniques.

Certora Prover

The Certora Prover is a formal verification tool that analyzes smart contracts against specifications written in Certora Verification Language (CVL). The Prover detects cases where the source code deviates from the specification.

Certora Prover uses a combination of static analysis and SMT (Satisfiability Modulo Theories). SMT is a technique in formal verification that combines aspects of theorem proving and symbolic execution.

Installation:
Install the following dependencies to use Certora Prover:

- Python3.8.16 or newer.
- Java Development Kit (JDK) 11 or newer.
- Solidity compiler (v0.5 and up).

Next, to install Certora Prover run:

```
pip3 install certora-cli
```

How to use:
The Certora Prover CLI requires an API key to interact with the cloud service where the results of verfication can be visualized.

Certora also provides a free demo cloud service with examples to see how the tool works.

Specifications for Certora Prover are written in Certora Verification Language (CVL), starting with the rule keyword.

Specifications in CVL are saved with a .spec file extension.

A simple specification that ensures that the balance of an ERC20 holder and recipient in a transfer call are updated correctly can be written like this:

```
methods{
    // declaration of the balanceOf functionfunction balanceOf(address)
external returns(uint256) envfree;
}

// the parameters will be fuzzed
rule transferSpec(address holder, address recipient, uint256 amount)
  description "Verifying that the balances of the ERC20 sender and
recipient are updated correctly after a successful transfer"
{
  // Initialize the variables `holderBalBefore`,
`recipientBalBefore`,// `holderBalAfter`, and
`recipientBalAfter`.uint256 holderBalBefore = balanceOf(holder);
  uint256 recipientBalBefore = balanceOf(recipient);

  // Create an environment object `e` which contains global variables
and set the `msg.sender` property to// `holder`.
  env e;
  require e.msg.sender == holder;

  // Call the `transfer` function with the specified `recipient` and
`amount`// arguments.
  transfer(e, recipient, amount);

  // Update the variables `holderBalAfter` and `recipientBalAfter` to
reflect// the new balances of `holder` and `recipient`.uint256
holderBalAfter = balanceOf(holder);
  uint256 recipientBalAfter = balanceOf(recipient);

  // Assert that the balance of `holder` is decreased by the amount of
the// transfer.assert holderBalAfter == assert_uint256(holderBalBefore
- amount);
```

```
  // Assert that the balance of `recipient` is increased by the amount
of the// transfer.assert recipientBalAfter ==
assert_uint256(recipientBalBefore + amount);
}
```

The CVL specification has a similar syntax to solidity as seen in the code above.
The 'env' keyword encompasses solidity global variables and the 'envfree' keyword indicates that the method or function does not rely on a global variable.

In CVL 2, the results of all arithmetic operators (and not bitwise operators) is of type mathint (an integer type that does not overflow or underflow in CVL), regardless of the input types. This is why we typecast the right hand side of our assertions to uint256 with assert_uint256().

We will use this specification to formally verify the vulnerable ERC20 contract below.

```
// SPDX-License-Identifier: MITpragma solidity ^0.8.0;contract
VulnerableERC20 {
    // A lot has been abstracted away from the contract, leaving only
the ERC20 transfer function.mapping(address => uint256) balances;
    event Transfer(address indexed _from, address indexed _to, uint256
_value);

    function transfer(address _to,
        uint256 _value
    ) public returns (bool success) {
        require(balances[msg.sender] >= _value);

        uint fromBalance = balances[msg.sender];
        uint toBalance = balances[_to];

        balances[msg.sender] = fromBalance - _value;
        balances[_to] = toBalance + _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function balanceOf(address _owner) public view returns (uint256
balance) {
        return balances[_owner];
    }
}
```

We will create a VulnerableERC20.sol and ERC20.spec file for the contract and CVL specifications respectively.

(Note that an API key which is set as an environment variable in the terminal (export CERTORAKEY=<premium_key>) is required to access all the features of the prover, you can chat with the Certora team on discord on how to setup or use a demo version with less computational effort by just running the command below).

We can verify that the contract matches the specs by running:

```
certoraRun VulnerableERC20.sol:VulnerableERC20 --verify
VulnerableERC20:ERC20.spec
```

This is the result from the run.

We can see from the result that the prover found a bug that violates our specification.

Click to see the full report.

From the call trace, we see that the "holderBalBefore" is 11, and the amount of token transferred is 9.

"holderBalAfter" is meant to be 11 - 9 = 2 after the transfer. Because the balances are actually being updated wrongly, this deduction does not work if both the token sender and receiver is the same address as show in the result.

However, this deduction would only work if the token sender and reciever are different addresses.

But since both addresses are the same, we are supposed to have the same token balance before after the transfer (that is, 11). This is because the balance of the address would be deducted as the sender and added back as the receiver which the ERC20 token standard allows for.

To fix this, we can replace the transfer function in the vulnerable contract with the code below.

```
function transfer(address _to,
    uint256 _value
 ) public returns (bool success) {
    require(balances[msg.sender] >= _value);
    // update balances correctly
    balances[msg.sender] -= _value;
    balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
 }
```

This would update the balances correctly.

We have to also update our CVL specification to catch thee exception where the address of the token sender and receiver are the same.

```
methods{
  function balanceOf(address) external returns(uint256) envfree;
}

rule transferSpec(address holder, address recipient, uint256 amount)
  description "Verifying that the balances of the ERC20 sender and
recipient are updated correctly after a successful transfer"
{
  // Initialize the variables `holderBalBefore`,
`recipientBalBefore`,// `holderBalAfter`, and
`recipientBalAfter`.uint256 holderBalBefore = balanceOf(holder);
  uint256 recipientBalBefore = balanceOf(recipient);

  // Create an environment object `e` and set the `msg.sender` property
to// `holder`.
  env e;
  require e.msg.sender == holder;

  // Call the `transfer` function with the specified `recipient` and
`amount`// arguments.
  transfer(e, recipient, amount);

  // Update the variables `holderBalAfter` and `recipientBalAfter` to
reflect// the new balances of `holder` and `recipient`.uint256
holderBalAfter = balanceOf(holder);
  uint256 recipientBalAfter = balanceOf(recipient);

  // If the `holder` and `recipient` are the same, // assert that the
balance of `holder` is the same after the transfer.if(holder ==
recipient){
    assert holderBalAfter == holderBalBefore;
  } else{
    // Assert that the balance of `holder` is decreased by the amount
of the// transfer.assert holderBalAfter ==
assert_uint256(holderBalBefore - amount);

    // Assert that the balance of `recipient` is increased by the
amount of the// transfer.assert recipientBalAfter ==
assert_uint256(recipientBalBefore + amount);
```

```
        }
    }
```

This time we have added an “if statement” if(holder == recipient) to check for this exception.

Rerun the test with, certoraRun VulnerableERC20.sol:VulnerableERC20 --verify VulnerableERC20:ERC20.spec.

The test passes this time and no rule failed.

Click to see the full report.

The Certora team also has some helpful VSCode extensions you can install to run verification jobs right from the IDE.
To see more examples of how CVL specifications are written and used to formally verify smart contracts, visit the demo website for Certora Prover or watch the Getting started tutorial.

Strengths

- **Certora uses automatic formal verification to compile smart contract code into mathematical formulas. This is a precise and effective method for finding bugs through formal verification.**
- **Unlike fuzzing, Certora Prover can start testing the smart contract at an arbritray state, thus increasing test coverage.**
- **Variables and functions can be fuzzed in Certora Prover.**
- **Certora Prover supports declaring smart contract invariants and other helpful methods.**
- **It does not generate false positives.**
- **The Certora Verification Language is easy to use and write.**
- **Supports CI integration.**
- **Certora CVL has similar syntax to solidity, hence learning it is not difficult.**
- **The combination of SMT and static analysis reduces the complexity of SMTs and also eliminates the false positives of static analysis.**

Limitations

- **Formal verification is automateed with Certora Prover and this is computationally expensive.**
- **With Certora Prover and most formal verification tools, there can be false negatives because the tools cannot test outside the specifications. Any part of the code not covered in the specification would not be tested.**

- Certora Prover is a commercial tool, and is not totally to use.

Solidity SMTChecker

Solidity SMTChecker is a built formal verification module in the solidity compiler (solc).

SMTChecker is based on SMT (Satisfiability Modulo Theories) and Horn solving.
It encodes program logic from solidity into SMT statements and uses that to check for assertion failures.

It interprets require and assert statements as formal specification. It takes a require statements as assumptions (assumes it to be true) and tries to prove that the conditions inside assert statements are always true.

Installation:
SMTChecker comes as a part of the solidity compiler (solc).
See installation process from the solidity documentation.

How to use:
The SMTChecker can be activated in a solidity file by add the pragma experimental SMTChecker; pragma directive.
To verify the code with the BMC engine, we can run this command on the solidity file:

```
solc <solidity-file> --model-checker-engine bmc
```

The SMTChecker has several options, run solc to view them.
Verifying the Quadratic contract with SMTChecker
We will be verifying the Quadratic contract from our foundry invariant article with SMTChecker and BMC engine.

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma experimental SMTChecker;
pragma solidity ^0.8.6;contract Quadratic {
    bool public ok = true;

    function notOkay(int x) external {
        if ((x - 11111) * (x - 11113) < 0) {
            ok = false;
        }

        // we "expect" this will never get executedassert(ok == true);
    }
}
```

Save the code as Quadratic.sol and run, solc Quadratic.sol --model-checker-engine bmc.

We get this output:

SMTChecker uses z3 solver by default and it was able to find the correct number that breaks the assertion.

Strengths

- **It comes directly with the solidity compiler and does not need any complex configuration to use.**
- **SMTChecker supports two model checking engines, Bounded Model Checker (BMC) and Constrained Horn Clauses (CHC). BMC explores a finite number of states or execution paths of a program (smart contracts in this case) and checks if any of the paths violates an assertion, while CHC explores all paths.**
- **SMTChecker is faster to use than most formal verification tools when using the BMC engine.**
- **It can check for 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.**

Limitations

- **SMTChecker cannot check all possible errors with BMC, since it only explores a finite path and might miss some cases where asserions may fail.**
- **The CHC engine is computationally expensive to use and can be very slow.**
- **SMTChecker is an experimental feature and is still under development.**
- **False positives may occur due to abstraction and external function calls.**
- **Contract state invariants cannot be automatically deduced by the SMT checker.**

Halmos

Halmos is an open source symbolic execution tool for Ethereum smart contracts developed the a16z team.

It uses symbolic methods and bounded model checking (checking code correctness within a limited number of steps), which allows it to find bugs that would not be found by traditional testing methods.

Installation:
To install halmos, run:

```
pip install halmos
```

Verify that halmos is installed with:

```
halmos -h
```

How to use:
To perform a symbolic execution test with halmos, run halmos command in a solidity project.

We will be testing the contract from the dapptools section of this article.

```
// SPDX-License-Identifier: GPL-3.0-or-later
pragma solidity ^0.8.0;
contract Code {
    uint256 private password;

    constructor() payable {
        require(msg.value >= 1 ether);
        password = 25;
    }

    function withdraw(uint256 password_) public {
        if (password_ == password) {
            (bool success, ) = msg.sender.call{value:
address(this).balance}(
                ""
            );
            require(success, "failed to send");
        }
    }

    function balance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Instead of fuzzing the contract, we will test the contract symbolically with halmos to see if it finds the password to the contract.

We run halmos in the project directory and get this result.

From our result, we can see halmos was able to find the password of the contract through symbolic test.

The “paths” from the test result indicates how many execution paths or sequences halmos can take and how many it actually took. In this case, it took one out of five paths.

The “bounds” represent upper and lower limits to input variables for the contract during the test process.

Strengths

- It is easy to use and it is good in finding bugs.
- Halmos uses bounded symbolic execution to avoid the halting problem. Bounded symbolic execution limits the number of times that a loop can be executed. This allows Halmos to explore all possible paths through a program, even if the program contains unbounded loops.
- It is constantly being improved by the developers.

Limitations

- Halmos is a new tool and is still under development. As such, it may not be able to find all bugs in smart contracts.
- Halmos does not support all smart contract opcodes. This means that it may not be able to verify the correctness of contracts that use certain features.
- Halmos can be slow, especially when testing complex smart contracts. This is because it has to explore all possible execution paths of the contract, which can be a very large number.
- It is not perfect and might give false negatives.
- Bounded symbolic execution is not a perfect solution. It can miss errors that occur in loops that are executed more than the specified bound.

HEVM

Hevm is an implementation of the Ethereum Virtual Machine in Haskell made solely for finding bugs in and verifying the correctness of Ethereum smart contracts using symbolic execution.

Installation:
Hevm comes with dapptols, you can check out the dapptools section to see installation method.

How to use:
Hevm has several commands, but symbolic testing with hevm begins with the hevm symbolic keyword.

You can symbolically execute deployed smart contracts using and an rpc or test local contracts. Hevm will try to find execution paths where assert statements break. contract.

Verifying the Quadratic contract with HEVM.
We will demo how to test locally with hevm.
We would verify the Quadratic contract again.

```
// SPDX-License-Identifier: GPL-3.0-or-laterpragma solidity
^0.8.6;contract Quadratic {
    bool public ok = true;

    function notOkay(int x) external {
        if ((x - 11111) * (x - 11113) < 0) {
            ok = false;
        }

        assert(ok == true);
    }
}
```

The goal is to find an integer that will make “ok” false.

We have to compile the code to bytecode before we can test the assertion with hevm.
To this this run the following command:

```
solc --bin-runtime -o <destination/path> <path/to/Quadratic.sol> --
overwrite
```

(Replace <destination/path> and <path/to/Quadratic.sol> with the appropriate file paths).

This generate a “Quadratic.bin-runtime” file which contains the runtime bytecode of the smart contract.

Next, run this command in the same path as the bytecode to perform a symbolic test.

```
 hevm symbolic --code $(<Quadratic.bin-runtime) --sig "notOkay(int x)"
--solver cvc4
```

(The --solver flag indicates which smt solver to use, in this case CVC4. With the default being Z3 theorem prover).

We get this output when we run the command.

Hevm was able to get the correct value in one try, unlike foundry fuzzer which needed bounded values to get the number.

We can verify that the number is actually correct and less than zero using chisel.

We can also add the --show-tree and --smttimeout <INTEGER> flags to print branches explored during the execution in a tree view and increase or decrease the timeout of the test.

To see all available commands for symbolic testing, run hevm symbolic -h.

Strengths

- **Hevm can be used to perform unit test and debug smart contracts.**
- **It is high perfomant and effective in symbolic test.**
- **Hevm also uses smt solvers like z3 and cvc4 to prove the correctness of code.**

Limitations

- **Hevm can be complex to use, as it requires a lot of command line commands.**
- **It is slow when testing large codebases.**

Manticore(symbolic execution)

Manticore is a symbolic execution tool for analyzing smart contracts and binaries. It can explore possible states of a program, generate inputs, and detect errors.
Manticore supports Ethereum smart contracts, Linux ELF binaries, and WASM modules.

Installation:
It is recommended to intall manticore in a python virtual environment with python >=3.8 <=3.9.x. This is because manticore uses other versions of some dependencies that might affect other tools in your computer.

To create a virtual environment, run the following commands:

```
pip install venv
python3 -m venv venv
```

```
source venv/bin/activate
```

Next install manticore in the virtual environment.

```
pip install manticore
```

How to use:
To use manticore and start a symbolic test on the specified contract, run:

```
manticore <path/to/file.sol>
```

If there's an unsat return message, it means that the solver was unable to find a solution to the given formula.

Incase you run into error like this:

```
1. Downgrade the protobuf package to 3.20.x or lower.
2. Set PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python (but this will use
pure-Python parsing and will be much slower).
```

Run this commands to resolve this (macOS/Linux):

```
pip install protobuf==3.20.0
export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python

# Rerun manticore
manticore
```

Strengths

- **Manticore is a powerful symbolic execution engine that can reason about different execution paths, states of a program, and sequences of transactions. This makes it effective in finding bugs that would be difficult or impossible to find with traditional testing methods (e.g, manual testing and static analysis).**
- **It has a flexible state management lifecycle that allows it to be used to analyze a different kinds of software like Ethereum smart contracts, Linux ELF binaries, and WASM modules.**

Limitations

- **Manticore can have conflicting dependencies with other installed CLI tools which might prevent it from running. To avoid this, it is recommended to install and use Manticore in a virtual environment.**
- **It is memory intensive and can be slow to run on large or complex software.**

4. Visualisation Tools

Smart contract visualization tools are very helpful to auditors and developers, they help visualize the structure and control flow of smart contracts.
They can help in visualizing contract layouts, inheritances, internal and external calls.

This can help to better understand a smart contract and give better documentations.

We write about some of them below.

4a. Surya by ConsenSys

Surya is a tool by the ConSenSys team for visualizing the structure of smart contract. With surya we can flatten contracts, see inheritance tree, see function traces and generate graphs, which aids in manual inspection of smart contracts.
Installation:
Surya requires graphviz to generate graphs as pictures.
Install graphviz:
Linux:

```
sudo apt-get install graphviz
```

macOS:

```
brew install graphviz
```

Install surya:

```
npm install -g surya
```

How to use:
We will use surya to see the function trace, inheritance tree and generate pictoral graphs for the solidity files below.

- **Unlock.sol**

```
// SPDX-License-Identifier: UNLICENSEDpragma solidity ^0.8.9;contract
Unlock {
    uint public unlockTime;
    address payable public owner;
}
```

- **Lock.sol**

```
import "./Unlock.sol";

contract Lock is Unlock {
    event Withdrawal(uint amount, uint when);

    constructor(uint _unlockTime) payable {
        require(
            block.timestamp < _unlockTime,
            "Unlock time should be in the future"
        );
```

```
        unlockTime = _unlockTime;
        owner = payable(msg.sender);
    }

    function withdraw() public {
        require(block.timestamp >= unlockTime, "You can't withdraw
yet");
        require(msg.sender == owner, "You aren't the owner");

        emit Withdrawal(address(this).balance, block.timestamp);

        owner.transfer(address(this).balance);
    }
}
```

We place both files in the same folder as the Lock contract inherits the Unlock contract.

Generating graph

To generate graph for the Lock contract, run this command in the directory:

```
surya graph Lock.sol | dot -Tpng > Lock.png
```

A Lock.png image is created and looks like this.

We can run the command without | dot -Tpng > Lock.png, this will print out the raw format in the terminal.

Generating inheritance graph

To generate

```
surya inheritance Lock.sol | dot -Tpng > Lock-Inheritance.png
```

Checking function trace

To see the withraw function trace in the Lock contract, run:

```
surya ftrace Lock::withdraw all Lock.sol -i
```

We get the trace as shown below:The withdraw function accesses the address of the Lock contract and also makes an external payable call tp the owner address.

Flattening the Lock contract

We can flatten the Lock contract to get all the inherited contracts in one file with.

```
surya flatten Lock.sol > Lock_flatten.sol
```

This will generate a flattened solidity file named Lock_flatten.sol.

Getting contract description

With the describe flag, we can show contract structure.

```
surya describe Lock.sol
```

This gives us a summary of the Lock contract:

Now run surya to see all available commands.

Strengths

- **Surya is very efficient in manual auditing processes and can help auditors better understand smart contracts by visualizing the structure of smart contracts.**
- **It works better with graphviz tool and can be used to generate helpful call trace and inheritance graphs, which is helpful for documentations.**

Limitations

- **it requires the graphvis tool to generate graphs and this comes with extra commands.**
- **Surya is less intuitive to use than the solidity visual auditor tool which also makes use of it internally.**

4b. Solidity visual auditor

Solidity visual auditor a.k.a solidity visual developer is a VSCode extension by tintinweb that supports syntax highlighting for solidity code, detects and highlights some security issues.

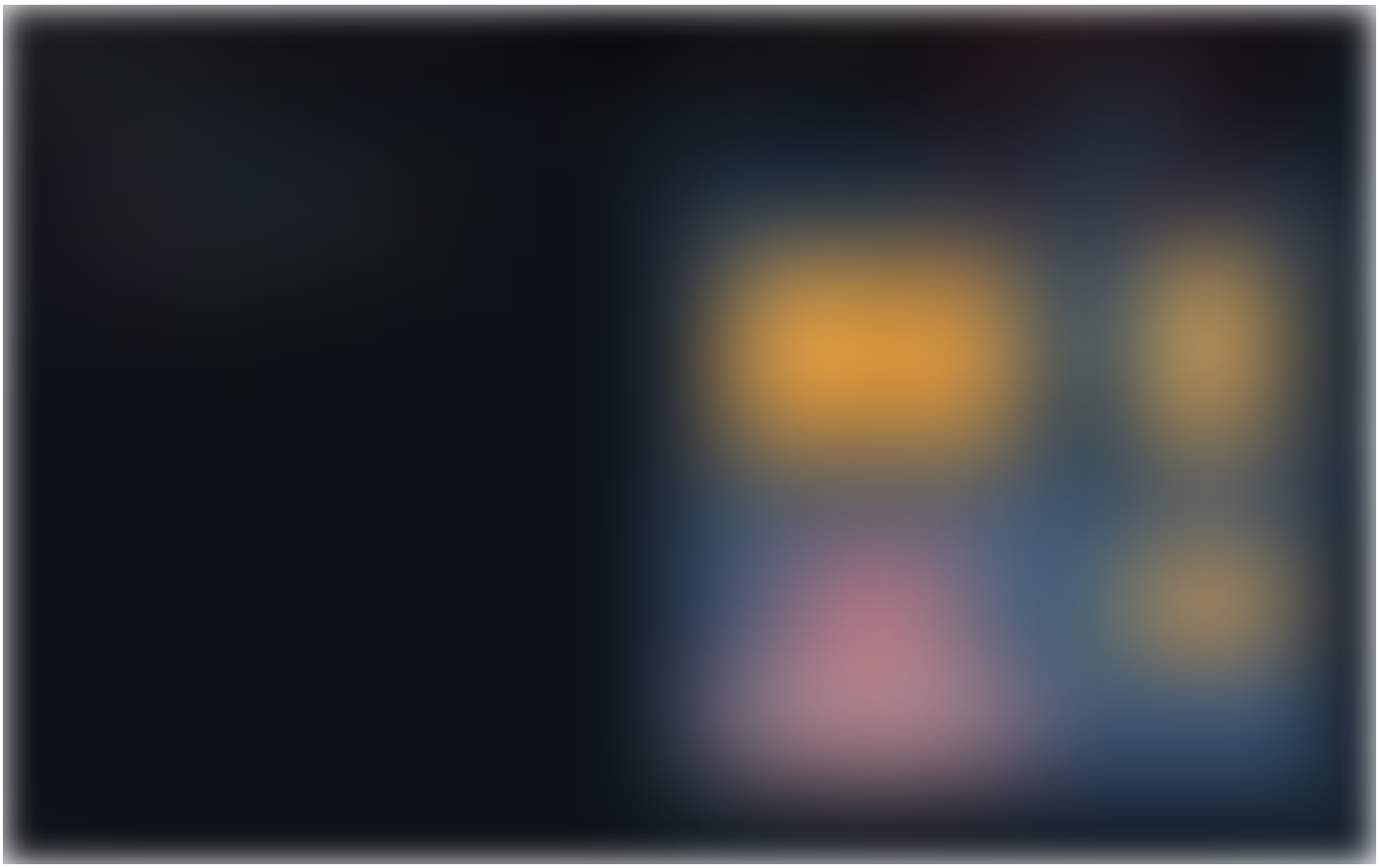
It uses surya internally and can aslo be used to flatten solidity files, generate inheritance graphs, call graphs even reports.

Installation:
To install, download the extension from the VSCode extension marketplace.

How to use:
After installation, the extension icon can be found at the primary side bar of VSCode which shows an outline view for contracts present in the project, as shown in the image below.
Open any solidity project and tap on the extension's icon.

We can now effortlessly interact with surya and generate graphs by selection the solidity file.

Here’s a graph generated with the extension, we can also select what type of graph we want.



Strengths

- The extension supports semantic highlighting of state variables, including their type and visibility.
- It can detect state variable shadowing and some bad patterns which can lead to bugs.

- **Highlights access modifiers, such as public and private, to help you understand the scope of variables and functions.**
- **Comes with a number of themes to customize the look and feel of the VSCode IDE.**
- **Highlights unsafe calls, external calls, and other interactions, which can help you identify potential security vulnerabilities.**

Limitations

- **Some of the extension's features, such as ftrace (function tracing), may not perform well for large codebases.**
- **The outline view may not always update.**

Learn more

See our state-of-the art smart contract developer bootcamp to learn more about using these tools, and many more advanced Solidity development subjects.