FCAT FIDELITY CENTER for APPLIED TECHNOLOGY℠

SMART CONTRACT SECURITY

By: Meenakshi Singh

May, 2023

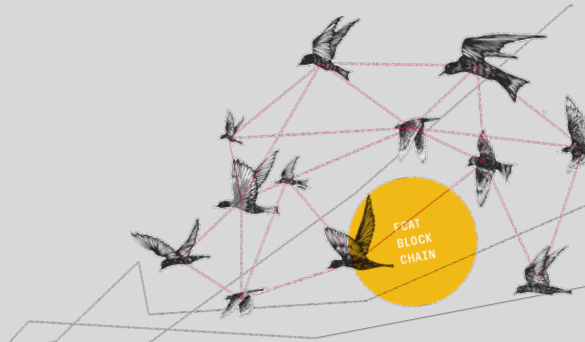
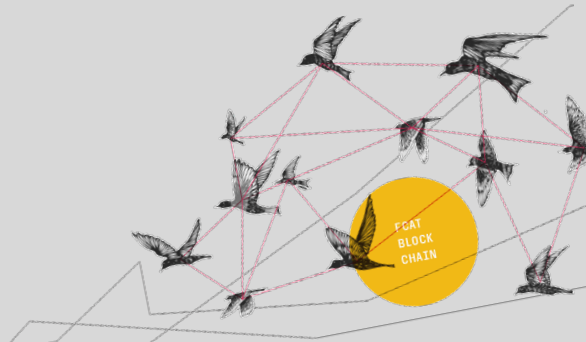
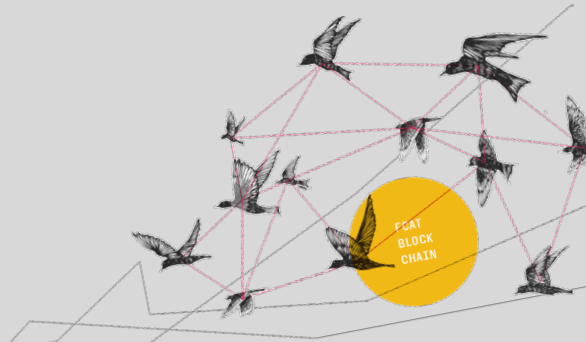


TABLE OF CONTENTS



EXECUTIVE SUMMARY

To mitigate the risks of financial loss and fraud, smart contract developers must prioritize smart contract security best practices from the outset. While numerous tools are available in the space, promising to identify vulnerabilities and propose fixes, there is no one-size-fits-all solution. It is crucial to select tools with broad coverage, easy integration, and generic applicability to reinforce security and reliability throughout the smart contract's development lifecycle.



1. INTRODUCTION

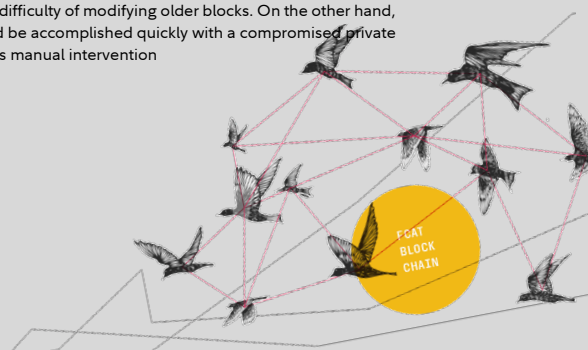
Blockchains are practically immutable*, meaning that once a transaction is recorded on the blockchain, it cannot be altered. Smart contracts are self-executing in the sense that they can automatically enforce the terms of the agreement without the need for human intervention. That said, once they are deployed on the blockchain, they automatically execute when a specific trigger condition is met. In a trustless environment where human bias in decision making is absent, smart contracts can establish trust among parties. This also makes them a target for malicious attacks. Therefore, it is important to thoroughly analyze smart contracts to ensure that they have zero security vulnerabilities before deploying them on the blockchain, as a single vulnerability could result in significant financial losses.

Smart contract programming requires a different engineering mindset compared to traditional programming. The cost of failure can be high and updating deployed contracts can be challenging. Therefore, it's essential for developers to defend against known vulnerabilities while staying up to date with changes in the security landscape. Various analysis techniques and tools have been developed over time to ensure that smart contracts are safe and secure.

This paper covers common developer oversights that can result in security breaches in smart contracts, followed by a systematic review of different open-sourced Ethereum smart contract analysis tools and techniques. The objective of this review is to assist developers in selecting the best toolset to perform security analysis for their smart contracts. In total, 25 tools were analyzed. In the interest of brevity, we offer a comparative analysis of only a subset of more conspicuous tools. Obsolete tools, which are no longer supported by the community or newer versions of solidity, were not considered.

Finally, this paper describes our recommended approach towards the creation of a project agnostic, simple and easy-to-use, Continuous-Development-Continuous-Testing (CD/CT) pipeline by employing the tools selected in our review. Adherence to a CD/CT development paradigm should significantly reduce the chance of bugs in the development of smart contracts, promote adherence to code quality and best practices during development and testing phases, and provide greater confidence in deploying them on a public blockchain.

* Immutability of Blockchains can be challenged in certain situations, such as in public chains with 51% attacks or private chains with less rigorous consensus protocols. That said, it's important to note that in Bitcoin, immutability is ensured by chaining blocks with hashes, which increases the difficulty of modifying older blocks. On the other hand, in Fabric, given that block chaining does not contribute to immutability hardness and tampering with the ledger could be accomplished quickly with a compromised private key, the design of Fabric does not permit ledger forks and any such occurrence is a signal of an issue that necessitates manual intervention



2. COMMON DEVELOPER OVERSIGHTS

In Q1 2022 alone, the DeFi industry lost over $1.6 billion [1] due to exploits, surpassing the total amount stolen in 2020 and 2021 combined. Some protocols were hacked because of simple code errors, while others were vulnerable due to inefficient contract logic or incorrect calculations. The Smart Contract Weakness Classification Registry [11] compiles a list of smart contract vulnerabilities with test cases for developers' reference. Let's go over some of the most common developer oversights [4] that have resulted in the loss of funds.

2.1. Integer Arithmetic Errors [2,5]

Smart contracts use integers to represent numbers, as they do not support floating-point numbers. This means that numerical values must be expressed in smaller units, such as 18 decimal places, to maintain precision. A simple example of this concept is using cents instead of dollars, as $0.5 cannot be represented using integers. When integers reach their maximum value, they loop back to the minimum value, which can cause issues with overflow. Overflows are shown in in the figures below, for both unsigned and signed integers. As soon as the stored value reaches 0xff...fff, adding 1 will reset the storage to minimum value, i.e., 0x0.

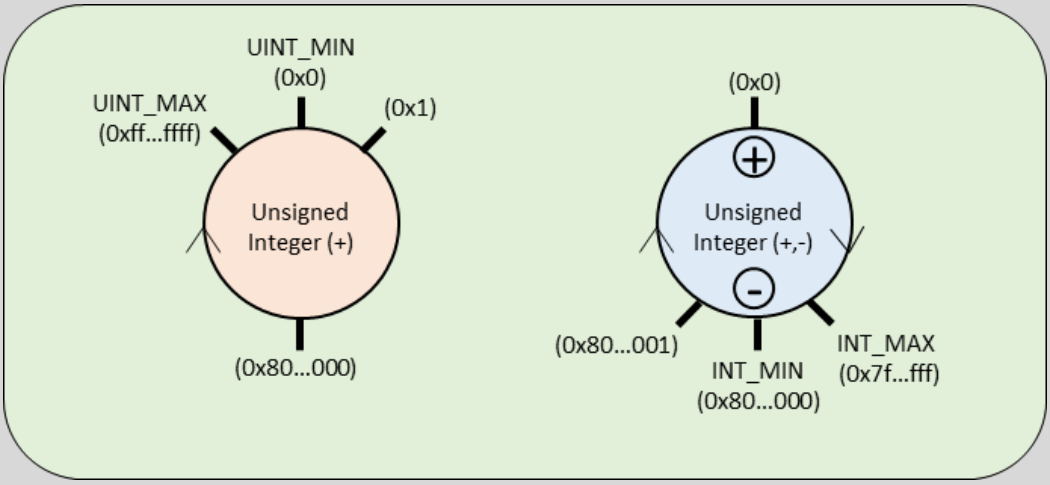


Figure 1: Integer Overflows

To avoid these mistakes, using the latest version of Solidity (solc v8 onwards) is crucial as it comes with built-in support to handle overflows. Utilizing libraries like Open Zeppelin's safe math can also be beneficial. Paying close attention to the order of operations when working with integers in smart contracts is also key. Let's look at an example code:

```solidity
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.12;

contract demonstrateTimelock {

    mapping(address => uint) public lockTime;
    mapping(address => uint) public totalBalance;


    function depositFunds() external payable {
        totalBalance[msg.sender] += msg.value;
        lockTime[msg.sender] = block.timestamp + 1 weeks;
    }

    function withdrawFunds() public {
        require(totalBalance[msg.sender] > 0, "Insufficient funds in the account");
        require(block.timestamp > lockTime[msg.sender], "TimeLock not expired, try later!");

        uint amount = totalBalance[msg.sender];
        totalBalance[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }

    function increaseLockTime(uint _timeInSec) public {
        lockTime[msg.sender] += _timeInSec;
    }
}
```

Figure 2: Integer Overflow Code Sample

This contract is meant to lock the user's Ether for a week and is vulnerable to overflow/underflow attacks because the *withdrawFunds()* function logic is dependent on a uint. A malicious actor could write code to deposit Ether to the contract and then withdraw a larger amount before the lock period, *lockTime* expires, by *"overflowing"* (essentially, resetting) the lock time function, *increaseLockTime*, to zero.

2.2. Block Gas Limit Vulnerabilities [4,5]

The block gas limit ensures that the amount of gas consumed by transactions in blocks is kept within a certain limit, to ensure that the transactions can be executed. However, if data is stored in dynamic arrays and then accessed through loops, the transaction may not have enough gas and be undone. This can occur when the number of elements in the array becomes large. This problem is particularly dangerous because it can go unnoticed during testing since test data is often smaller. Contracts with this issue may pass unit tests and appear to work well with a small number of users, but they can fail when the amount of data increases. It is not uncommon for funds to become irretrievable if loops are used to make payments. In these cases, one needs to keep track of how far the processing is done, and be able to resume from that point, as in the following example:

```
struct ReceiverAddress {
    address addr;
    uint256 holding;
}

ReceiverAddress[] receiverAddresses;
uint256 nextReceiverAddressIndex;

function payOutFunction() {
    uint256 i = nextReceiverAddressIndex;
    while (i < receiverAddresses.length && msg.gas > 200000) {
      receiverAddresses[i].addr.send(receiverAddresses[i].holding);
      i++;
    }
    nextReceiverAddressIndex = i;
}
```

Figure 3: Block Gas Limit Vulnerability Code Sample

Developers need to make sure that other transactions don't have any critical dependency on the *payOutFunction()* and are processed successfully while waiting for its next iteration.

2.3. Reentrancy [3,4]

A Reentrancy Attack is an improper enforcement of behavioral workflow. Ethereum smart contracts can call and utilize the code of external contracts and, in many cases, send ether to external user addresses. Cybercriminals can steal these external calls and force the contract to execute a call back to itself (using a malicious fallback function). The execution of the code *"re-enters"* the contract recursively before the contract can update its state and drains the funds. To prevent this, one can use the built-in transfer() function when sending ether to an external contract as it only sends 2300 gas, which isn't enough for the destination address/contract to re-enter the sending contract, along with ensuring that all state-change logics execute before ether is sent out of the contract. For example:

```
// INSECURE CODE - REENTRANCY THREAT
mapping (address => uint) private userFunds;

function withdrawFunds() public {
    uint amount = userFunds[msg.sender];
    require(msg.sender.call.value(amount)());
    userFunds[msg.sender] = 0;
}
```

Figure 4: Reentrancy Vulnerability Code Sample

The above code resets the value of the *userFunds* back to zero after the *withdrawFunds()* function is called. This makes the contracts vulnerable to reentrancy attacks because it sends Ether to the user before updating the user's balance, giving the receiver an opportunity to call the contract again in its fallback function before the balance is updated.

2.4. Frontrunning [4]

Frontrunning can be defined as overtaking an unconfirmed transaction. Blockchains are designed to be transparent which means that all unconfirmed transactions are visible in the mempool before they are included in a block by a miner. A malicious party could monitor mempool transactions for their content and overtake them by paying a higher transaction fee. For example, let's say, Alice grants Bob permission to use 100 tokens. Later, Alice decides to revoke this permission and attempts to reduce Bob's allocation to 50 tokens. However, Bob, who is monitoring the transaction closely, quickly creates his own transaction spending the original 100 tokens and uses a higher gas price, which prioritizes his transaction over Alice's. This can result in Bob being able to access 150 tokens.

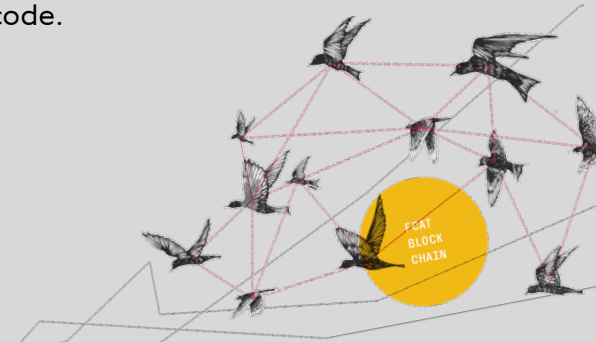
One way to prevent front-running is to set a maximum limit on the gas price in the smart contract, making it impossible for users to change the gas price. However, this approach only protects against typical users, as miners can still bypass the limit. Additionally, it can be challenging for miners to target a specific block, so in many cases, this may not be a concern. Another method to prevent front-running is to use a commit-and-reveal scheme. This involves sending an encrypted transaction first (the commit phase), and then later, sending another transaction that decrypts the information from the first transaction (the reveal phase). This approach makes it impossible for both typical users and miners to perform front-running attacks.

3. SMART CONTRACT TESTING AND COMMON VULNERABILITY ANALYSIS TECHNIQUES

Smart contract testing involves thoroughly analyzing and evaluating the source code during development to ensure its quality. By testing a smart contract, potential bugs and vulnerabilities can be identified and addressed, reducing the risk of costly software errors. Strategies for testing Ethereum smart contracts can be classified into two broad categories: manual testing and automated testing [6].

3.1. Manual Testing

Manual testing of smart contracts is a process where humans conduct testing by executing steps manually. Code audits, where developers and/or auditors go over every line of contract code, are an example of manual testing. However, manual testing requires an attacker mindset and a significant investment of time, money, and effort. It can also be prone to human errors, such as missing a data type deep down in a complex function call-stack, which may lead to overflows. Despite these limitations, manual code audits leverage the unique capabilities of humans, such as understanding and analyzing code in a way that automated tools cannot, resulting in a more thorough and nuanced examination of the code.



3.2. Automated Testing

Automated testing involves using tools, which can execute repeated tests, to carry out scripted testing of smart contracts. Automated testing is efficient, uses fewer resources, and promises higher levels of coverage than manual analysis. Automated tests can also be configured with test data, allowing them to compare predicted behaviors with actual results.

We will concentrate the following sections towards the three most common automated analysis techniques: Static and Dynamic analysis, upon which most popular tools are built.

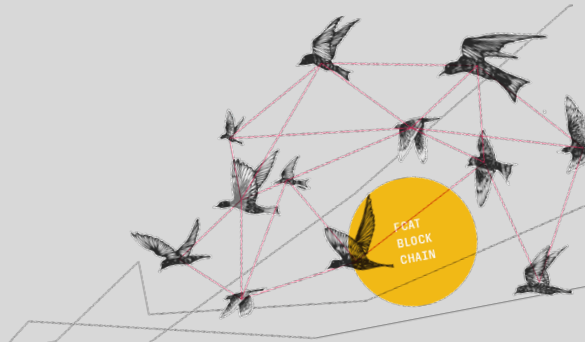
3.2.1. Static Analysis [6]

Static code analysis is a method of debugging by examining source code before a program is executed. It's done by analyzing a set of code against a set (or multiple sets) of coding rules and is often used interchangeably, along with *source code analysis*. Static analyzers can detect common vulnerabilities in Ethereum smart contracts and aid compliance with best practices.

3.2.2. Dynamic Analysis [6]

Dynamic analysis requires executing the smart contract in a runtime environment to observe contract behaviors during execution. There are multiple techniques to achieve this; here are the two which stand out:

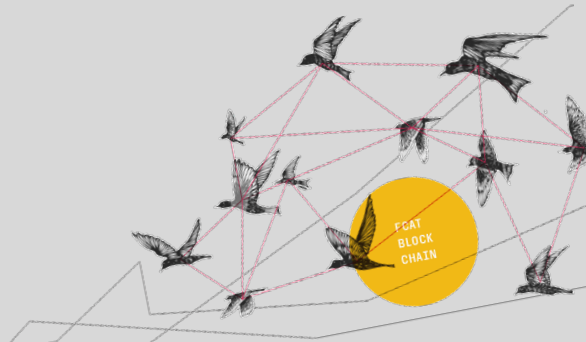
1. **Symbolic Execution** explores multiple paths that a program could take under different inputs or symbolic values. The main goal of this technique is to explore as many different program paths as possible, generate a set of concrete input values to execute each generated path, and finally to check for the presence of errors. It allows creation of high coverage test suites and provides developers with concrete inputs that trigger bugs. Major challenges in effectively performing symbolic execution include constraint solving, combinations of which can easily go over hundreds of variables, compromising its efficiency and decidability, and space explosion, which prevents the engine from exhaustively exploring all possible paths within a reasonable amount of time due to the exponential increase in the number of execution paths.
2. **Fuzzing** is another popular example of dynamic analysis. During fuzz testing, a fuzzer feeds the smart contract with malformed and invalid data and monitors how the contract responds to those inputs. Like any program, smart contracts rely on inputs provided by users to execute functions and sending incorrect input values to a smart contract can cause resource leaks, crashes, or worse, and lead to unintended code execution.



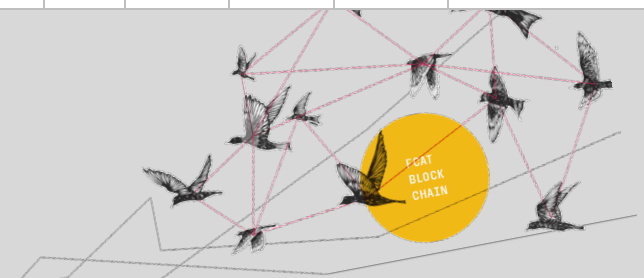
4. CONTINUOUS DEVELOPMENT/CONTINUOUS TESTING (CD/CT) SMART CONTRACT VULNERABILITY TESTING PIPELINE

We examined over 25 open-sourced smart contract security analysis tools, but for brevity, we have provided a comparison of a few notable tools and excluded those that are no longer relevant. We grouped the tools based on the analysis techniques they use, and then evaluated them against a set of well-reasoned parameters. Through this approach, we have identified a selection of tools that we recommend for detecting vulnerabilities in smart contracts.

Based on our analysis, we concluded that all open-source tools have limitations in terms of their capabilities in one way or another. Hence, the most effective way to create an efficient and well-rounded testing tool suite, capable of observing known bug patterns in an exhaustive manner, is likely picking the tool that performs each popular analysis technique the best. As we saw, the most common smart contract analysis techniques are static analysis, symbolic analysis, and fuzzing, which led us to pick Slither, Mythril and sFuzz, respectively, for our CD/CT smart contract test suite. Detailed analysis of tools is presented in table below:



Name	Analysis Technique	Package Manger/OS	Optimization*	False Positives*	Average Execution Time*	Solidity Version Support	CI/CD Integration	Reporting Text*	No SWC vulnerabilities	Code Optimization & Issues	Notes
Slither Developed by Trail of Bits	Static	pip3, Linux + Windows	High	Low	Fast (< 1 sec per contract)	>=0.4	Yes	Good	78	Yes	Offers APIs, has extendible core to cover additional vulnerabilities, comes with flattener. Can detect conformance to various ERCs (ERC20, ERC721) and checks for contracts using the delegate call proxy pattern for upgradable contracts. Extensive configurations provides numerous options to run analytics on a local file, Etherscan, and AST file. Options for different printers, detectors, path filtering, as well as Triage mode (users can mark false positives to be excluded from consecutive runs). Does not have detectors for integer overflow and underflow.
SmartCheck Developed by SmartDec	Static	NPM	Med	High	Average	>=0.6.0	No	Good (not color coded)	20	Yes (less thorough)	First security tool written in Vyper (Python+Solidity). Can identify functional violations and operational issues like runtime problems and bad performance. Doesn't perform numerical analysis and therefore did not detect integer overflow/ underflow. Points out potential out out-of-gas problems.
Manticore Developed by Trail of Bits	Symbolic	pip (python > 3.7)	Low	Low	Very Slow	>=0.4	No	Nice	13	No	Besides Ethereum contracts, it can also scan x86/64, ARM binaries. Good code coverage - probability of spotting a vulnerability is high. Can't find business logic vulnerabilities. Auto-generates inputs for triggering unique code paths and records instruction-level execution traces for crashes. Exposes its analysis engine via Python API



Securify2.0 Developed by Ethereum Foundation + Chain Security	Symbolic and Context Sensitive static analysis	pip3	Medium	High	Slow	>=0.5.8	No	Poor	38	Yes	Only supports flat contracts (no imports), must use flattener tools before using this. Lacks detection of numerical properties violation (overflow/ underflow) and reachability. Since it requires higher solidity version so 3 slither patterns and 2 SWC vulnerabilities are covered by solidity compiler itself.
Mythril Developed by Consensys	Symbolic Execution	pip3 only Linux	Med	Low / Med	Slow (Depends on max recursion depth)	>=0.4	Yes	Good (detailed, not color coded)	15	No	Supports the analysis of multiple Blockchains other than Ethereum that make use of EVM and only require the EVM bytecode to analyze the smart contract. Mythril emulates contract execution, stores all execution branches, and strives to reach a "dangerous" contract state trying different parameter combinations and possible options. Performs numerical analysis, detects bad randomness, out of bounds array access and unprotected ether withdrawal. Fails to detect transaction order dependency, outdated compiler version and bad solidity coding best practices. Does not provide many control options such as path filtering and Triage mode. Slow due to trade-off between recursion depth and execution time. Maximum recursion depth can be controlled; default value is 12.
Echidna Developed by Trail of Bits	Fuzzing	pip	Med	Low	Average	>=0.4	Yes	Nice	14	No	Generates inputs tailored to your actual code, powered by Slither to extract useful information before the fuzzing campaign. Source code integration to identify covered lines of code. Reports maximum gas usage. Property based fuzzer. Overall, isn't very practical due to bugs and setup challenges.
ContractFuzzer Developed by Bo Jiang	Fuzzing (property based)	Docker	Low	High	Slow	N/A	No	Poor	7	No	Leverages test oracles to execute the contract with multiple and different inputs to try to trigger a strange behaviour to spot vulnerabilities. Fuzzer engine does not use any feedback to improve the test suite.

sFuzz Developed by Tai D. Nguyen	Fuzzing (Brute-Force based)	CMake	Med	Low/ Med	Fast	>=0.4	No	Poor	9	No	Leverages test oracles to execute the contract with multiple and different inputs to try to trigger a strange behaviour to spot vulnerabilities. Fuzzer engine does not use any feedback to improve the test suite. sFuzz requires only the EVM bytecode to fuzz smart contracts. Performs numerical analysis, detects numerical overflows and underflows. Uses evolutionary AFL fuzzer method (popular for c/c++ programs) with a lightweight multi-objective feedback-guided adaptive strategy that targets those difficult-to-cover branches. Based on a feedback-guided adaptive fuzzing technique which transforms the test generation problem into an optimization problem and uses feedback as an objective function in solving the optimization problem. sFuzz = AFT + Smart Contract + lightweight SBST. Broadly, it has 3 main components: runner, libfuzzer and liboracles. Can complement other symbolic execution tools to enhance code coverage of the fuzzer. Effective in achieving high code coverage. sFuzz is not property based, less scope for customization and randomization of inputs based on business logic.
Fuzz testing in Foundry	Fuzzying (property based)	Foundry Binaries	Unknown	Variable (depends on properties)	Unknown	N/A	Yes	Poor	Unknown	No	Foundry provides property based fuzzy testing for solidity based smart contracts as a way of testing general behaviours as opposed to isolated scenarios. 2. "runs" refers to the number of scenarios the fuzzer tested. By default, the fuzzer will generate 256 scenarios, however, this can be configured using the FOUNDRY_FUZZ_RUNS environment variable. 3. "μ" (Greek letter mu) is the mean gas used across all fuzz runs. 4. "~" (tilde) is the median gas used across all fuzz runs.

Table 1: Comparative Analysis of Ethereum Smart Contract Vulnerability Testing Tools [7], [8], [9], [10]

** The data in columns 'Optimizations,' 'False-Positive Rates,' and 'Average Execution Times' are rated in relation to the performance data recorded for Slither.*

Key for reporting text:

Notes	Output color coding by severity	Reference to SWC Registry	Examples & recommendations to fix
Poor	No	No	No
Nice	No	Yes	Yes
Good	Yes	Yes	Yes

Table 2: Metrics for Poor | Nice | Good rating for reporting text generated by different tools.

** If integration with a CI/CD pipeline is not mentioned in the documentation of the tool, then 'No' is reported. However, all tools can be containerized using Docker and integrated into a CI/CD pipeline.*

Note: Our current security pipeline engages Slither and Mythril to perform smart contract vulnerability analysis. While working with sFuzz, we faced some technical challenges which could not be resolved and our pull request to resolve the same is still pending for review by the community. In the meantime, we are also looking into Diligence Fuzzing, a tool by Consensys, as an alternative. In this paper, we cover only open-source tools. As Diligence Fuzzing is subscription based, it was kept out of our current analysis scope.

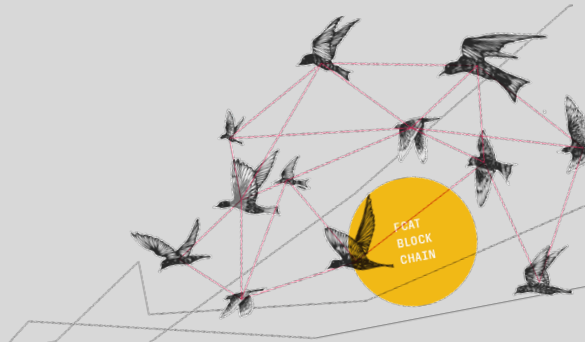
As most of these tools come with platform specific dependencies, we chose to create our CD/CT pipeline using Docker. The Dockerfile utilizes a rust-based image and installs the tools and their required dependencies with baseline configuration settings.

5. PROJECT SETUP AND SAMPLE RUN COMMANDS

To setup our own CD/CT pipeline using docker, we can leverage latest rust image and extend it to install all the required dependencies, as enumerated in the list below:

1. Python
2. Slither
3. Mythril
4. Solidity Version Manager (SVM)

Following is the Dockerfile with all the installation commands:



```
FROM rust:latest

USER root

# Define and provide proxy arg
ARG http_proxy
ARG https_proxy
ENV http_proxy=${http_proxy}
ENV https_proxy=${https_proxy}

# Use rust nightly
RUN rustup default nightly

# Install Python
RUN apt update
RUN apt install -y python3 python3-pip libleveldb-dev

# Install Slither
RUN pip3 install slither-analyzer

# Install Mythril
RUN pip3 install mythril

# Install SVM
RUN cargo install --git https://github.com/roynalnaruto/svm-rs.git

# Create security dir
RUN mkdir /security
WORKDIR /security

# Copy configs/test scripts, make unix execute, copy configs
COPY ./ .
RUN sed -i -e 's/\r$//' run-vulnerability-tests.sh
RUN chmod +x run-vulnerability-tests.sh

# Set default env vars
ENV MYTHRIL_MAX_DEPTH=5 \
    MYTHRIL_EXECUTION_TIMEOUT=300 \
    PROJECT_HOME="/app" \
    CONTRACTS_LOCATION="contracts" \
    SOLC_VERSION="0.8.16" \
    RUN_SLITHER="true" \
    SLITHER_CONFIG="/security/slither.config.json" \
    RUN_MYTHRIL="true" \
    MYTHRIL_CONFIG="/security/mythril.config.json" \
    MYTHRIL_MAX_DEPTH=5 \
    MYTHRIL_EXECUTION_TIMEOUT=300

ENTRYPOINT /security/run-vulnerability-tests.sh
```

Figure 5: Sample security container Dockerfile.

If you need any other project specific tools, you can extend this base image and install them in your Dockerfile. However, this image should be sufficient for setting up and running our CD/CT security pipeline.

Next, you need a *docker-compose.yml* file, to customize the CD/CT suite to your project's specific needs. The compose file will pull and build the security container based upon the default configuration settings, which can be modified as needed. A sample configuration file is shown in the image below:

In case if you are using your custom image, you can change the image reference here.

```yaml
version: "3"

services:
  solidity-security:
    stdin_open: true
    tty: true
    image: rust:latest
    container_name: solidity-security
    build:
      context: ./docker/security
      dockerfile: Dockerfile
      network: host
      args:
        http_proxy: //your-http-proxy-setting//
        https_proxy: //your-https-proxy-setting//
    volumes:
      # Volume containing code
      - ./:/app
    network_mode: "host"
    entrypoint: "/bin/bash"
    environment:
      PROJECT_HOME: "/app" # Location of project/volume containing code. Other locations are relative to this
      CONTRACTS_LOCATION: "contracts" # Location of contracts in project
      SOLC_VERSION: "0.8.13" # Version of solc to compile code with
      RUN_SLITHER: "true"
      SLITHER_TRIAGE_MODE: "false"
      SLITHER_CONFIG: "/security/slither.config.json" # Location of slither config relative to project
      RUN_MYTHRIL: "true"
      MYTHRIL_CONFIG: "/security/mythril.config.json" # Location of mythril config rleative to project
      MYTHRIL_MAX_DEPTH: 5 # default_value is 12
      MYTHRIL_EXECUTION_TIMEOUT: 300 # value of timeout is in seconds
      container_name: solidity-security
```

Figure 6: Sample fact-solidity-security container docker-compose file.

If you have your own Dockerfile, you can provide it in the `*services.solidity-security.build.{context | dockerfile}*` tag. Proxy settings can be done using the `*services.solidity-security.build.args.{http_proxy | https_proxy}*` tags. The compose file is configured to read the tool configuration files from our `*security*` folder.

The compose file associates the entire project directory structure as an external volume (`*services.solidity-security.volumes*`) to the security container. This enables the modifications in the files to be instantly reflected within the container without having to restart it. This aides the

developers by instantly verifying their changes to ensure that they did not accidentally introduce a potential bug in the system.

As evident from the image above, the default directory for solidity contracts is `*contracts*` and solidity version set is `*0.8.12*`. This can be set to any other version, which the prebuilt SVM will install and use to compile contracts.

Developers can toggle between running Slither and Mythril along with an option to run both. As Mythril is backed by a symbolic execution engine, it takes a bit longer to run. One complete run on our set of smart contracts lasted 20 minutes. Supporting configurations for both Slither and Mythril such as modes, depths, execution timeouts are kept in the compose file to allow easy reconfigurations without building the containers again and again. The container can easily be integrated with the Jenkins environment with its build status configured to the outputs of individual tools.

Following are the snapshots of our *slither.config.json* and *mythril.config.json* files respectively. The configuration files provide individual parameters to report output folders, solidity compiler remapping's, filter paths and report preferences etc.

```
{
  "json": "./security/reports/slither/slither-output.json",
  "solc_remaps": ["@openzeppelin/=node_modules/@openzeppelin/"],
  "exclude_informational": true,
  "exclude_low": false,
  "exclude_medium": false,
  "exclude_high": false,
  "disable_color": false,
  "filter_paths": "(ERC-1404/|openzeppelin/)",
  "legacy_ast": false
}
```

Figure 7: Sample Slither config file.

Mythril, doesn't provide many options to finetune report preferences as provided by Slither. You can only configure the solidity compiler remappings in its configuration file. Other required settings are done as run time parameters while executing the Mythril command tool.

```
{
    "remappings": [ "@openzeppelin/=node_modules/@openzeppelin/" ]
}
```

Figure 8: Sample Mythril config file.

Finally, we have our script – run-vulnerability-tests.sh. The script, as shown in the image below, first reads the solidity compiler version, installs it if it isn't available, sets the directory for publishing the reports (`*security/reports/{mythril | slither}*`), and then runs Slither and Mythril,

taking the configurations provided in their respective config files as well as environment variables set in the docker compose file.

```
#!/bin/sh

# Install relative solc version
if [ -n "${SOLC_VERSION}" ]; then
  echo "Installing solc version ${SOLC_VERSION}"
  # Remove then install to garuntee version is present
  svm remove ${SOLC_VERSION} >/dev/null
  svm install ${SOLC_VERSION} || exit 1
else
  echo "Please set SOLC_VERSION env var to desired solc version."
  exit 1
fi

# Move into project directory
cd ${PROJECT_HOME}

# Directory for security related reports
SECURITY_DIR="./security/reports"

echo "----------Run Slither and generate report------------------"
if [ "${RUN_SLITHER}" = true ]; then
  SLITHER_OUTPUT_DIR="${SECURITY_DIR}/slither"
  # Create slither directory if not existant
  if [ ! -d ${SLITHER_OUTPUT_DIR} ]; then
    mkdir -p ${SLITHER_OUTPUT_DIR}
  fi

  # Remove previous file if existing, create new slither output
  SLITHER_OUTPUT_JSON="${SLITHER_OUTPUT_DIR}/slither-output.json"
  rm -rf ${SLITHER_OUTPUT_JSON}
  slither "${CONTRACTS_LOCATION}" --config-file "${SLITHER_CONFIG}"
else
  echo "Skipping slither testing."
fi;

echo "----------Run Mythril and generate report------------------"
if [ "${RUN_MYTHRIL}" = true ]; then
  MYTHRIL_OUTPUT_DIR="${SECURITY_DIR}/mythril"
  # Create slither directory if not existant
  if [ ! -d ${MYTHRIL_OUTPUT_DIR} ]; then
    mkdir -p ${MYTHRIL_OUTPUT_DIR}
  fi

  # Remove previous file if existing, create new mythril output
  MYTHRIL_OUTPUT_JSON="${MYTHRIL_OUTPUT_DIR}/mythril-output.json"
  find ${CONTRACTS_LOCATION} -name '*.sol' | xargs myth analyze --solc-json ${MYTHRIL_CONFIG} -o jsonv2  > ${MYTHRIL_OUTPUT_JSON}
else
  echo "Skipping mythril testing."cd
fi;
```

Figure 9: Sample script file to run Slither and Mythril.

The security container can be built and started in the background with the help of the following command:

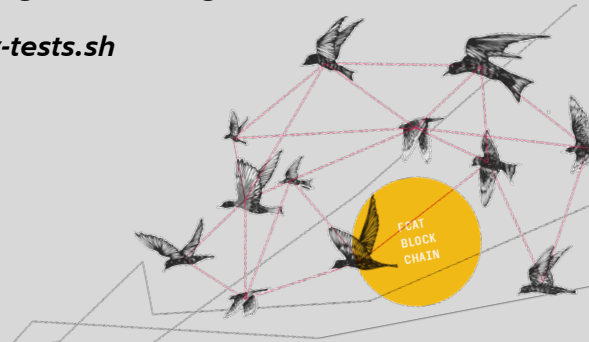
docker-compose -f docker-compose-<example>.yml up -d

And the containers can be stopped/torn down with the following command:

docker-compose -f docker-compose-<example>.yml down

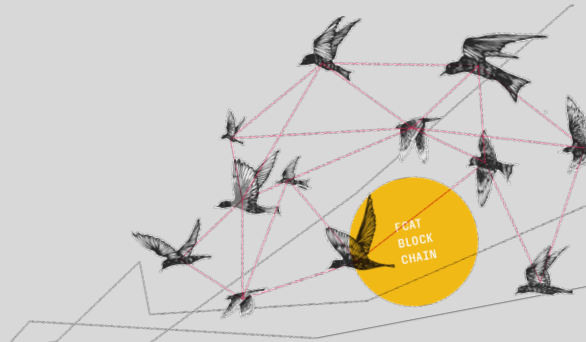
Once the container is running, the reports can be generated using the following:

docker exec solidity-security /security/run-vulnerability-tests.sh



6. CONCLUSION

Securing smart contracts begins with thoroughly reviewing each line of code. It is crucial to formally verify contracts after they have been developed and tested but implementing effective security measures during the development process can help prevent bugs and make informed design choices. The CD/CT security pipeline described in this paper can facilitate this by allowing developers to continuously test changes without having to continually rebuild the security container. This system, which is based on Docker, is also easily extensible, saving time and reducing the need for reconfiguration. Additionally, the reports can be integrated into Jenkins jobs for more efficient builds.



7. REFERENCES

1. *More than $1.6 billion exploited from DeFi so far in 2022.* (n.d.). Cointelegraph.
2. 5 Most Common Smart Contract Vulnerabilities. (2020). https://medium.com/cryptronics/the-5-most-common-smart-contract-vulnerabilities-738de4fae3ba
3. Chibuzor, M. (2022, June 17). *Smart contract development: Common mistakes to avoid.* LogRocket Blog.
4. *Known Attacks - Ethereum Smart Contract Best Practices*. (n.d.). Ethereum-Contract-Security-Techniques-And-Tips.readthedocs.io.
5. 16 Solidity Hacks/Vulnerabilities, Fixes and Real-World Examples. (2018). https://medium.com/hackernoon/hackpedia-16-solidity-hacks-vulnerabilities-their-fixes-and-real-world-examples-f3210eba5148
6. Ethereum Smart Contracts Testing. (2022). https://ethereum.org/az/developers/docs/smart-contracts/testing/
7. *Security Tools - Ethereum Smart Contract Best Practices*. (n.d.). Ethereum-Contract-Security-Techniques-And-Tips.readthedocs.io
8. Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., & Lee, H.-N. (2022). Ethereum Smart Contract Analysis Tools: A Systematic Review. *IEEE Access*, 1–1.
9. *The Landscape of Solidity Smart Contract Security Tools in 2020.* (2020, December 23). Kleros.
10. Oualid, Z., & Oualid, Z. (2022, April 26). *Top 10 solidity smart contract audit tools*. Get Secure World.
11. *Overview · Smart Contract Weakness Classification and Test Cases.* (n.d.). Swcregistry.io.

1079263.1.0