National College of Ireland

# Solidity Smart Contract Testing with Static Analysis Tools

MSc Research Project
Cybersecurity

Senan Behan
Student ID: x20167601

School of Computing
National College of Ireland

Supervisor: Mr. Vikas Sahni

**National College of Ireland**

**MSc Project Submission Sheet**

**School of Computing**

**Student Name:** ……Senan Behan……

**Student ID:** ……x20167601……..……

**Programme:** ……MSCCYBETOP…… **Year:** ……2022……...

**Module:** ……Research Project ……..……

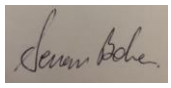
**Supervisor:** ……Mr. Vikas Sahni……..……

**Submission Due Date:** ……15 August 2022……..……

**Project Title:** ……Solidity Smart Contract Testing with Static Analysis Tools ……..……

**Word Count:** ……6847…… **Page Count**……20……..……..

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

**Signature:** ……Senan Behan…… 

**Date:** ……14/08/2022……

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST**

| | |
|---|---|
| Attach a completed copy of this sheet to each project (including multiple copies) | □ |
| **Attach a Moodle submission receipt of the online project submission,** to each project (including multiple copies). | □ |
| **You must ensure that you retain a HARD COPY of the project**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. | □ |

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

| **Office Use Only** | |
|---|---|
| Signature: | |
| Date: | |
| Penalty Applied (if applicable): | |

# Solidity Smart Contract Testing with Static Analysis Tools

Senan Behan
x20167601

**Abstract**

Smart contract development is often overlooked in security terms and the consequences of vulnerable smart contracts embedded within a Blockchain can lead to current and future unforeseen negative consequences. Solidity smart contracts are a rapidly developing area within Blockchain technology. Several static analysis tools have been developed to assist in the secure creation of smart contracts, and datasets are provided to facilitate testing of tools.

This report describes the results of testing the Static Analysis tools, Osiris, Oyente and Slither against Solidity generated smart contracts which contained documented vulnerabilities, sourced from Smart Contract Weakness Classification and Test Cases (SWC) registry and SmartBugs repository. The Docker static analysis tool images can be utilised in testing to enhance security in smart contracts. The findings in this report demonstrated the dominance of Slither testing tool in scanning and detecting vulnerabilities, however False Negatives were present. The experiment also highlighted the issue of vulnerability classification in datasets and re-classification of the dataset smart contracts for vulnerabilities is required. This investigation demonstrated that Docker proved to be an effective means of testing the tools.

Keywords: Slither, Osiris, Oyente, SmartBugs, SWC Registry, smart contracts, vulnerabilities, Solidity, Static Analysis, testing.

# 1 Introduction

After the Colonial Pipeline attack in the USA, the White House signed in an Executive Order which marked a new dawn in standard setting for organisations and testing of their cyber and IT infrastructures. The European Union brought in the Cybersecurity Act 2019 Regulation (EU) 2019/881 which sets out certification standards for organisations which are utilising best practice cybersecurity measures.

Blockchain is becoming part of critical structures, and Smart Contracts, programming scripts with functions and rules [1], allow autonomous actions/transactions once deployed on the blockchain. Industrial Standards Organisations have published standards relating to smart contracts to improve development and security, standards such as the International Standardization Organization, ISO 23455:2019 and the European Telecommunications Standards Institute (ETSI), GS PDL 011, both of which define a smart contract "…as a computer program stored in a distributed ledger, wherein the outcome of executed programs is recorded on the distributed ledger" [2].

Solidity based smart contracts designed for Ethereum are vulnerable as highlight by Chen et al [3], who stated that the programming for Ethereum is a new "…paradigm with DApps running on top of blockchains with many autonomous contracts…" resulting in undiscovered vulnerabilities which require time to discover and resolve. Bouichou, Mezroui, and Oualkadi [4] describe Solidity as "…loosely typed languages don't require the programmer to be specific…" resulting in vulnerabilities.

Static Analysis tools and smart contracts tested by researchers in the past, require revisiting, due to the development of new static analysis tools and their suitability, new vulnerabilities discovered and the challenge of understanding of vulnerability classification.

## 1.1 Research Question

*"How effective are different docker static analysis testing tools at detecting different vulnerabilities in Solidity generated Smart Contracts?"*

## 1.2 Objectives

In addressing the research question, this report is organised as follows: Section 2 is the Related Work, where a literature review is presented in the following key categories: Vulnerabilities of Smart Contracts, Types of Smart Contract, and Categorizing of Vulnerabilities and Analytic Tools / Security Tools. Section 3 is the Research Methodology section outlining the research, equipment and the techniques utilised, and the dataset gathered and processed. Section 4 describes the framework of the research, while section 5 elaborates on the final stages of the implementation, and section 6 discusses the evaluation of the research. Lastly, future works are discussed, and the conclusion is presented in section 7.

The research question posed is conducted on Docker Image Static Analysis tools Osiris, Oyente and Slither, noting that the three Docker Images. One previous study, Ren et al. [5] compared the above mentioned tools, focusing on runtime parameter. Testing in the present study is conducted against known smart contracts with known vulnerabilities, sourced form SWC registry [6] and Smartbugs [7]. All the known smart contract vulnerabilities listed in the SWC registry are tested without code alteration. Tests are conducted on smart contracts with either one or more vulnerabilities, noting the different levels of scanning capability of the tools.

The objective is to compare the static analytic tools against known vulnerabilities testing the static analysis tools for True Positive, False Positive, False Negative and True Negative detection to determine the efficiency the different static analysis tools.

# 2 Related Work

## 2.1 Background

Solidity, written in a high-level language, is a programming language for creating smart contracts, within an Ethereum Virtual Machine (EVM), where the EVM, acting as a sandbox for smart contracts in Ethereum [8] runs/complies the code.

Solidity [9] and Ethereum [10] are open source allowing many new developers access, bring collaboration and publication of readymade smart contracts. Solidity and the Ethereum platform has experienced a big demand [11] in 2022 noted by the increase of job vacancies. Despite the vulnerabilities of Solidity, 83% of developers [12] do not view security as a priority. For the above reason this report will focus on Solidity created smart contracts.

## 2.2 Vulnerabilities

Smart contracts vulnerabilities have led to a number of real world attacks include, inter alia, the DAO in 2016 [13], PoWH Coin Ponzi Scheme [14], the Parity MultiSig Wallet [15], The King of the Ether [13] and Coindash ICO attack [16] with vulnerabilities; re-entrancy, batch Transfer Overflow CVE-2018-10299, DELEGATECALL opcodes and DoS, Unchecked Return Values for Low Level Calls, Short Address respectively.

The level of security of a smart contract is reliant on the security of the programming language [3]. Emin Gün Sirer, a computer scientists who examined the DAO attack in the early stages, commented in a 2016 Article [17] that Solidity and Ethereum Virtual Machine are "…more designed for loose web code…" application, commenting that smart contracts should be treat in the same way as writing for nuclear code.

Smart contracts are deployed in EVM by way of complied bytecode. All rules and functions are programmed into the smart contract which invokes tasks/transactions as per programming. Once a smart contract is placed on the blockchain the smart contract is immutable, therefore any vulnerabilities or bugs in the coding will be permanent as the smart contract cannot be altered [13].

Praitheeshan et al. [1] detected 19 software and 16 Ethereum issues stating that vulnerabilities in smart contracts are a result of software code implemented, the Solidity language scripts and the immutability of blockchain. A number of challenges exist when utilising Solidity [1]: developers inexperience of this new technology, lack of best practices, immutability of smart contracts prevent updating of vulnerabilities.

### 2.2.1 Types of smart contract vulnerability

Vaibhav Saini listed 16 Solidity vulnerabilities in a 2018 Article [18], which are as follows:

- Re-entrancy
- Integer Overflow:
- Parity wallet attack:
- Party Wallet Attack II: Killed wallet
- Withdraw and not sent.
- Time draft (timestamp dependence)
- Separate public and private
- 51% Attack
- DelegateCall
- Default Visibility
- Default Visibility
- Entropy Illusion.
- Short Address
- Unchecked CALL Return Values
- Front Running (Race Condition)
- DDoS
- Constructors with Care
- Uninitialized storage Pointers
- Floating Points and Precision
- Tx.Origin Authentication

## 2.3 Tools

Static Analysis tools allow the examination of smart contracts for predefined common vulnerabilities without the need to deploy on the blockchain. The tools can examine the source code or compile the contract on a blockchain simulator (Virtual Machine) compiling the bytecode. While both paid and free static analysis tools for Solidity based smart contracts have been tested over the years, the research has not been exhausted. Further tools have yet to be examined and compared, and the varying metrics of the studies and datasets to be explained. Samreen and Alalfi [19] listed several static analysis tools which include: *Oyente, SmartCheck* which detects vulnerabilities patterns analysis, *Mythril* written in python executes the EVM bytecode "symbolically" disassembling the code, *Maian* detects multiple transactions executing the EVM Bytecode symbolically, *Securify* extracts from known vulnerabilities to create compliance and violation patterns and uses EVM bytecode as inputs, *Vandel* disassembles and decompiles bytecode, and *EthIR* which uses rule based Control Flow Graphs as input to a static analyser. Other publications mentioned static tools such as *Gigahorse* [20], *MadMax* [20], *Osiris* [21], *Slither* [22], *teEtherv* , *sComplier* [3].

## 2.4 Testing

Sharma et al. [12], in a very recent study, examined 29 developers use of smart contract testing tools, discovering a detection rate of vulnerabilities of 15% amongst inexperienced developers and 55% amongst experienced developers. Sharma et al. [12] observed that developers found tools difficult to implement, and they had to use different tools for different vulnerabilities. Furthermore, some tools had to be employed outside the complier. From the study [12] participant usage was as follows: 17% use static analysis tools, 14% use Truffle testing suite, 14% use Remix plugins, Myth was used by 7% and Slither by 3%. The study by Sharma et al. [12] was a review, and the authors did not conduct tool testing.

ISO TR 23455:2019 published in 2019 set standards for smart contract development including security of smart contracts. ETSI published in December 2021, an Industrial Specifications [23] on the requirements for Smart Contract's Architecture and Security, which gives guidance to testing.

## 2.5 Vulnerabilities of Smart Contracts

Saad et al. [24] examined attacks on smart contracts as part of Blockchain 2.0 focusing on Ethereum and Solidity, listing well-known attacks such as re-entrancy, over and under flow, replay, short address and reordering attacks, observing that attacks were due to poor programming or vulnerabilities in the program platform [24]. Saad et al. [24] stated that the flexibility in programming smart contracts contributed to vulnerabilities giving the example of re-entrancy attack which cannot occur on Bitcoin or Ripple. Saad et al [24] highlighted issues and an overview of fixes, but did not demonstrate the vulnerabilities through testing, nor did they suggest which testing tools to utilise.

Alkhalifah et al. [25] concentrated on the vulnerability re-entrancy, noting this vulnerability led to two attacks out of seven well documented incidents between 2016 and

2018, concluding that vulnerabilities in smart contracts were due to coding errors by the developer. Alkhalifah et al. [25], iterated that coding practices have not yet matured [26. p2], and referred the "code is law" concept, where once the smart contract is deployed on the Blockchain it is next to impossible to modify it.

The number of vulnerabilities varies depending on the research. Bouichou, Mezroui, and Oualkadi [4] listed 13 vulnerabilities, examining 8 real world smart contracts attacks and 12 testing tools, giving a good insight to the workings of the attacks, however, Bouichou et al. [4] limited the study to real world attacks. Chen et al [3] highlighted two security issues, firstly, permissionless i.e., allowing attackers in, and secondly, immutability which prevents vulnerability patching. Chen et al [3] viewed 40 vulnerabilities, 29 attacks with 51 defences, and focused on Ethereum, unlike Saad et al., Chen et al. discovered that 13 of the 25 Ethereum application layer vulnerabilities were not addressed/detected in their review, while observing that SmartCheck, performed well in discovering 10 out of 26 vulnerabilities. However, [3] the focus of the review was on defences and not on smart contract testing tools.

## 2.6 Categorizing of Vulnerabilities

Previous works tested smart contracts for known vulnerabilities (see Table 1 for examples), however, in many cases the vulnerability categorization was vague or the vulnerability in the smart contract was singularly classified [3], [19], [25]–[32]. Pervious papers did not conform to a standard classification of vulnerabilities and the labelling of vulnerabilities. In ten papers reviewed in this report [3], [5], [20], [26]–[32], which tested various tools, the dataset for the vulnerabilities was not consistent and nor were they fully comparable with each other. Each paper contained similarities in general labelling of the vulnerabilities, however, there was a degree of divergence with sampling specific vulnerabilities. This issue was highlighted by the ETSI in the 2021 published standard on smart contracts, stating that in testing, a well-structured approach was required and the test code should have clear indicators of error [23].

This view was iterated by Ren et al. [5]. In previous studies [31] [29] [28], the vulnerabilities selected for testing were generally categorised into headline vulnerabilities such as *re-entrancy* without expanding on the vulnerability mechanism, while other studies did elaborate on the mechanism, e.g., *fallback* function [20] or `call.value` [5]. The consequence of the above resulted in difficulty in comparing previous research due to labelling of datasets. Noting SWC registry lists 36 vulnerabilities of which exist subcategories (vulnerability mechanism) of the headline vulnerabilities, and Smartbugs provides the code associated with the headline vulnerability. SWC 106 labels unprotected "*selfdestruct*" functionality vulnerability, also known as a *suicide* vulnerability [20], under the headline vulnerability of "*access control"*. Papers [28] [20] [33] refer to "*suicide"* vulnerability, while papers [31] [20] [34] refer to this vulnerability as "*selfdestruct"*, and paper [5] refers to the vulnerability as an "*access control*" vulnerability. As observed, there is a degree of interchangeability concerning the abovementioned vulnerability.

As a further example, the vulnerability "mishandled exception" is also known as "Uncheck Call Return Value" [3], and Choi et al. [28] considers "*Gasless Send*" as a "*Mishandling Exception*". The above general labelling can cause confusion. The lack of a systematic labelling approach hinders comparison of tools.

Ren et al. [5] reported that the selection of the dataset (smart contracts with vulnerabilities) in many cases can skew particular results in favour of an author's tool which is subject to the research. Chen and colleagues argue that real world vulnerabilities are different to artificial injected vulnerabilities where the latter effects a "universal conclusion", highlighting the example of two contradictory results in terms of False Positive and False Negative for SmartCheck and Slither tools depending on the dataset utilised [5].

**Table 1: Vulnerability types and Previous Papers Published**

| **Publication** / **Vulnerability** | [27] | [3] | [20] | [26] | [28] | [5] | [29] | [30] | [31] | [32] |
|---|---|---|---|---|---|---|---|---|---|---|
| Re-entrancy | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Arithmetic | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| T Order Dependency | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ | | |
| Access Control /Suicidal | | | ✓ | ✓ | ✓ | ✓ | | | ✓ | ✓ |
| Unchecked LLC | ✓ | ✓ | | | | | | | ✓ | ✓ |
| Timestamp Dependency | ✓ | | | | | | ✓ | | | ✓ |
| Tx. Origin | ✓ | | | | ✓ | | | | ✓ | |
| Mishandling Exception | | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | | |
| DoS | | ✓ | ✓ | | | ✓ | | | | ✓ |
| Timestamp manipulation | | | | | | ✓ | | | | |

## 2.7 Analytic Tools / Security Tools

Static Analysis is the examination of code or programmes when the smart contract is not executed (non-runtime). Methods of Static Analysis include: decomplication, complying, run-based, CFG (Control Flow Graph and symbolic execution [1].

Ghaleb and Pattabiraman [27] compared their proposed SolidiFi tool against six static analysis tools using bug code snippets injected into "all possible locations" with an automated and systematic framework to evaluate the tools, with a focus on False Negatives. The injected smart contracts were examined by Oyente [35], Securify [15], Mythril [34], Slither, SmartCheck [30] and Manticore, checking for false negatives and undetected bugs, and presenting an explanation for the lack of detection. Osiris was not tested. The automated process of bug injection cannot be manually verified due to the large number of smart contracts injected, therefore comparison with other studies is difficult. The authors found Slither detected all re-entrancy vulnerabilities, however had a high False Positive return. Conducting the experiment with a smaller number of contracts with known vulnerabilities guarantees the vulnerability is known, and thus an improved analysis.

Di Angelo and Salzer [31] conducted a review of Ethereum smart contract static analysis tools arguing that little academic examination was previously conducted on the tools themselves, and that previous papers concentrated on methods, regardless of the tools' "provenance". Di Angelo and Salzer [31] examined 27 tools with the objective of creating a

guide for future developers. The report observed that some papers involved the examining of the author's own tool, giving rise to perceived impartiality when comparing the author's tool to others. Di Angelo and Salzer's [31], survey included Osiris, Oyente, finding that Osiris only detected Arithmetic issues and Oyente detected Re-entrancy, Timestamp and Trans Order Dependency. Slither was not included in the study.

Ji, Kim and Im [32] conducted testing of static analysis tools within docker containers testing against smart contract vulnerabilities in an attempt to demonstrate a proposed software tool to automate the evaluation static analysis tools. The proposed tool returned comparative performance indicators i.e., Recall, Precision, Accuracy and F1-Score for the static analysis tools. Slither and Oyente were among the tools test, however Osiris was not tested. A dataset of 237 smart contracts was utilised and which included re-entrancy, access control, unchecked LLC, and integer overflow. This paper concluded that tools which had a high True Positive has low precision and accuracy rates, recommending Slither and Oyente for re-entrancy, Slither for Uncheck LLC.

Perez and Livshits [26] argued that smart contract vulnerabilities may not always be exploitable. The authors reused the datasets of previous research, comprising of over 23,000 vulnerable contracts from 800,000 contracts. The authors settled on 6 vulnerabilities: "…re-entrancy (RE), unhandled exception (UE), locked Ether (LE), transaction order dependency (TO), integer overflows (IO) and unrestricted actions (UA)." Perez and Livshits [26] tested 9 static analysis tools including Oyente, which performed well recording that it detected RE,UE,TO and IO, however, the study did not include Slither nor Osiris.

Akca, Rajan, and Peng [36] examined their automated technique Solidity SolAnalyser, comparing it favourable to Oyente, Securify, Maian, SmartCheck and Mythril. This study involved generating approximately 13,000 mutated contracts from 1838 real contracts covering 8 different vulnerabilities. SolAnalyser utilised both static and dynamic analysis, combining *ContractAnalyser* and *ExecutionValidator*, creating SolAnalyser. This technique was compared to the abovementioned static analysis tools. Akca et al. [36] did observe that static analyses tools return a large number of False Positive. Of note, the study involved the authors own framework, giving rising the issues of impartiality.

Praitheeshan et al. [1] conducted a survey of previous works, listing a number of static analysis tools and the vulnerabilities detected. This paper comprehensively explained the main vulnerabilities; however, the vulnerabilities were limited to two real world attacks, namely DAO and Parity. The tested tools included, Oyente, Zeus, Vandel Ethir, Securify, Maian and Gasper which detected the headline vulnerabilities Re-entrancy, Exception handling, Transaction ordering, Block timestamp dependency, Call stack depth limitations, Integer over/under flow, Suicidal contract, Use of Origin, Unchecked and failed send, No restriction write, No restriction transfer and Greedy contracts. While this paper explored the headline vulnerabilities, the paper was a survey without testing the tools and did not report on the Recall, Precision or Accuracy of the tools in detail, only to state that Securify had less False Positives than Oyente and Securify stating was better at detecting *reentracy,* but the findings lacked detail.

Ren et al. [5] reported discrepancies between previous studies due to lack of a common framework for testing, common dataset and metrics to conduct correct comparison/evaluations based on runtime and evaluation metrics, testing Oyente, Osiris and Slither.[5] attempted to

create a baseline suite of tools and datasets for testing, selecting the dataset contracts form Etherscan (45,622) Solidify (9,369), CVE (124) and SWC (90). It was observed that the contract vulnerabilities gathered from Ehterscan API's were self-determined, and the authors conducted manual injection of vulnerabilities into the dataset from Solidify. While the authors created a baseline from a large dataset, the process above could have led to errors due to lack of third-party confirmation of the exitance and type of vulnerable. Furthermore, the study did not test the tool's ability to discover vulnerabilities.

Alkhalifah et al. [25] conducted testing on six "vulnerability-detection" tools to detect the re-entrancy vulnerability. The authors did discuss the short comings of the testing tools and their inability to discover new patterns in re-entrancy attack. The authors proposed a proof-of-concept solution, involving monitoring the difference between the contract balance and the total balance of all participants. Alkhalifah et al. [25] did not conduct testing on the different tools but confined their study to a proof of concept concerning re-entrancy vulnerability. The authors mentioned Oyente but did not test the tool.

Choi et al. [28] in their demonstration of SMARTIAN, a fuzzing testing tool, employed both statistical and dynamic analysis of smart contracts to contribute to the developing of the fuzzing testing. The static analysis was conducted to determine parameters for fuzzing and dynamic analysis conducted on dataflows for feedback. The experiment compared SMARTIAN against two fuzzer tools, ILF and sFuzz, and two symbolic executors Manticore and Mythril. The authors claimed SMARTIAN tested for 13 types of bugs, discovering 211 bugs from 500 real world contracts without excessive false positives. This study, while comprehensive, did not involve Slither, Oyente and Osiris. Furthermore, the authors were testing their own solution give rise to potential impartiality.

Chen et al. [3] outlined a number static tools which detected vulnerabilities referring to previous works, however, the paper did not conduct testing to confirm these claims.

# 3 Research Methodology

A framework comprising of static analysis smart contract tools running in Docker was established. The selected smart contract dataset was categorised based on vulnerabilities and tested by the static analysis tools within Docker. The preliminary results were examined to ensure the category/classification of the smart contract based on vulnerabilities corresponded with the results. If required, the smart contract maybe recorded as having an additional vulnerability assigned to that smart contract which was not originally classified according to the dataset source. The additional classification procedure is set out below. Additional classification may occur when a smart contract is labelled as having a certain vulnerability, but when tested by the tool, another vulnerability is detected. The "other vulnerability" maybe originally observed as a False Positive. On examination of the smart contract and supported by a detection by another static analysis testing tool, the vulnerability is deemed to exist and a return of True Positive, instead of False Positive, will be recorded, after which the results will be collated and compared.

The dataset of the smart contracts was copied into the Docker containers of the static analysis tool, and the tool was executed against the smart contracts. The results were returned and compared against the expected detection for that tool for each vulnerability, see Fig.1.

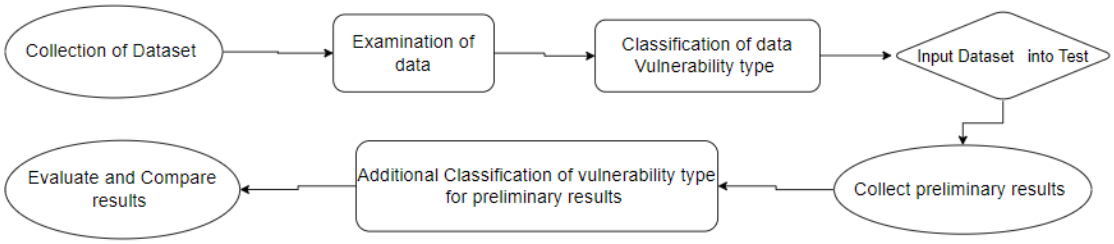


Fig.1. Workflow of Data Process

## 3.1 Equipment utilised

Equipment and software utilised as set out in the configuration manual

## 3.2 Data Selection

The dataset was selected from SWC Registry [6] and Smartbugs repository in GitHub [7].

## 3.3 Data Verification

Each individual smart contract was researched, and any attached commentary viewed to determine the type of vulnerability associated with the contract.

## 3.4 Data Integration

Each smart contract was individually selected and uploaded into VS Code to be complied and saved to a testing file. The contracts were viewed but were not altered from the original.

## 3.5 Data Testing

The experiment tested 207 smart contracts. Each static analysis tool is designed to test for certain vulnerabilities. The smart contracts selected form Smartbugs contained Re-entrancy, Access Control, Arithmetic and Unchecked Low-Level Call (Unchecked LLC) vulnerabilities and the dataset form SWC covers a greater range of vulnerabilities. The static analysis tools tested were not designed to detect all the vulnerabilities tested, a factor which was taken into consideration when determining detection rates. Smart contracts vulnerabilities which fall outside the scope of the test tools are considered neutral smart contracts for the purpose of the test and have an expected result of True Negative.

Solidity complier version of the testing tool effects the ability of the tools to read certain contracts. The solidity complier version in docker images Oyente and Osiris is pre-0.4.25, impacting on scanning a number of SWC smart contracts, while Slither complier version 0.4.25+ facilitated scanning all but a few of the contracts. Oyente read 72, Osiris read 69 and Slither read 185 contracts.

## 3.6 Testing tools

Three docker images based static analytical/ symbolic execution tools, Slither, Oyente and Osiris were tested in the experiment. The image of each tool was pulled from Docker Hub [37]

and executed in Docker as containers. The smart contracts are copied into the docker container of each tool and tested.

## 3.7 Evaluation

The smart contract testing result are separated into the following confusion matrix:
*True Positive (TP)*: The occurrence that the tool correctly detected a stated vulnerability.
*True Negative (NP)*: The occurrence that the tool correctly failed to detect a stated vulnerability as predicted.
*False Positive (FP):* The occurrence that the tool incorrectly detected a stated vulnerability.
*False Negative (FN):* The occurrence that the tool incorrectly failed to detect a stated vulnerability.

### 3.8.1 Detection of unlabelled vulnerability

This concerns two stages. Stage One, as illustrated in Table 2, compares the result i.e., the detection or non-detection with the documentation for the tool, ascertaining if the outcome is expected as per documentation. If the expected detection corresponds with the actual detection, the result will be a True Positive. However, if a detection for a vulnerability which is not listed or documented for that smart contract, then the process progresses to the next stage. Stage Two, as illustrated in Table 3, compares the result from another static analysis testing tool for the same smart contract. If two static analysis tools listed in Table 4 detect the same vulnerability in a smart contract which was not listed nor documented for that vulnerability, the result will be determined a True Positive, otherwise the result will be considered a False Positive.

### 3.8.1 Partial Detection vulnerability

Failing to detect a listed vulnerability while also detecting another unlisted vulnerability will be considered a False Negative for the undetected vulnerability, and the unlisted vulnerability detected will be further examined by a Stage Two process as mentioned above.

On occasion two smart contracts exist, where one contract contains the vulnerability, and the other contract is considered the "fixed" contract of the first, with no vulnerability. If a smart contract with a vulnerability is not detected i.e., False Negative, then the corresponding "fixed" contract will be not counted as a Ture Negative if the test indicates so, as it cannot be determined if the True Negative is as result of the "fix".

**Table 2: Expected detection by Static Analysis tool according to documentation for vulnerabilities in Smart Contract**

| Expected | Detection | Non-Detection | Future Examination | Classification |
|---|---|---|---|---|
| ✓ | ✓ | | | TP |
| ✓ | | ✓ | | TN |
| × | ✓ | | Revert (FP) | |
| × | | ✓ | | FN |

**Table 3: Revert -Vulnerability detected by an alternative Static Analysis Tool**

| Revert (FP) | Alternative Tool | Classification |
|---|---|---|
| | ✓ | TP |
| | × | FP |

**Table 4: Alternative Static Analysis Tool Combination**

| | Osiris | Oyente | Slither |
|---|---|---|---|
| Osiris | × | ✓ | ✓ |
| Oyente | ✓ | × | ✓ |
| Slither | ✓ | ✓ | × |

# 4 Design Specification

The collected dataset of smart contracts is compiled in VS Code then organised and saved into separate folders depending on the source of the smart contract, i.e., SWC or SmartBugs registry, and vulnerability type. Utilising a community edition docker engine installed on an Ubuntu 18.04.6 OS, the selected docker images for Smart Contract Static Analysis tools i.e., Oyente, Osiris and Slither are "pull" from the Docker Hub repository [37]. Within the Docker engine, set up docker containers for Oyente, Osiris and Slither from their retrospect images. The aforementioned smart contracts files will be copied into Oyente, Osiris and Slither docker containers via Docker, then execute the Static Analysis commands unique to each tool to test the individual smart contracts, see Fig.2.

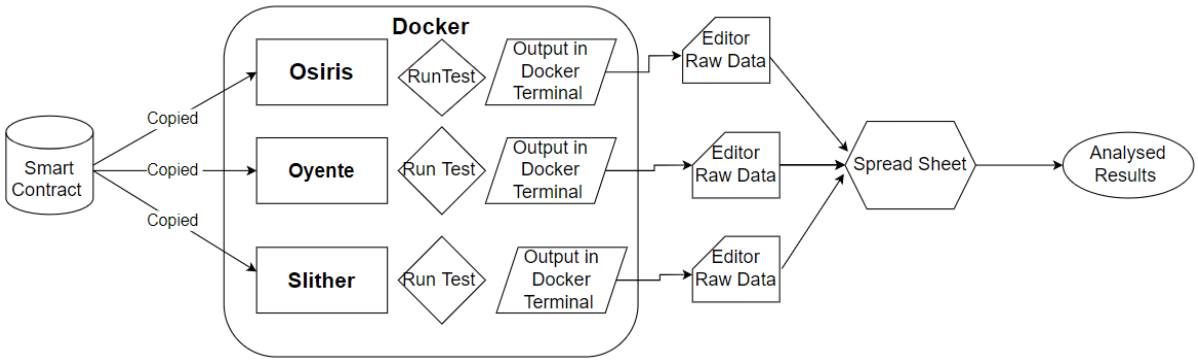



Fig.2. Design Specification of Docker hosting Static Analysis tools testing on Smart Contracts

# 5 Implementation

VS Code was utilised to compile and organise the dataset sourced form SWC registry and SmartBugs.

Docker Community Edition Engine was utilised to run Docker images, creating containers from Oyente, Osiris and Slither Static Analysis testing tools. The Docker images mentioned above scanned the smart contracts for vulnerabilities, and the results were outputted onto the docker terminal. The raw data results captured detection and non-detection both expected and unexpected of vulnerabilities and copied into an editor (Notepad ++). Comparison of the raw data results were made with the original documented smart contract and of the documented capability of the static analysis tools. Recorded results were entered a

spread sheet, including data handling errors. Statistical analysis was conducted creating a confusion matrix. Recall, Precision Accuracy and F1-Score were calculated, determining the capability and the predictability of each of the tools.

# 6 Evaluation

The 207 smart contracts tested by all tools resulted in both expect returns and some unexpected returns. Slither testing tool was the most comprehensive tool examining 185 smart contracts as Slither's solidity complier was able to read both newer and older contracts. Contracts with a complier 04.25.0 and earlier prove problematic for Oyente and Osiris. However, Oyente and Osiris read 72 and 69 smart contracts respectively. Oyente and Osiris have similar overall quantitative detection rate outcomes, noting a slight divergence on which contracts gave a particular result.

## 6.1 Dataset Classification

In determining if an unclassified vulnerability exists within a smart contract, the evaluation was conducted per Section 4.24 smart contracts were additionally classified as containing further vulnerabilities. When a new vulnerability was discovered originating from a particular classified smart contract, then that particular contract was additionally classified as containing a second vulnerable. For example, on examination, originally Unchecked LLC classified contracts were found to have an additional Re-entrancy vulnerability and 6 Arithmetic vulnerabilities as show in Table 5. Oyente and Osiris had a corelation of 1, discovering the same new vulnerabilities in contracts.

Table 5 shows the origin of newly discovered vulnerabilities within smart contracts. The column "SC Original Classification" is the original vulnerable type classified smart contract where the additional vulnerability was discovered. The top row "Additional Classification" represents the newly discovered vulnerability to be included in the experiment. A total of 24 smart contracts were re-classified with additional vulnerabilities. From examining previous publications, reclassification of SmartBugs smart contract was not conducted during testing.

**Table 5: Additional classified Smart Contract with Vulnerabilities**

| **Additional Classification** / **SC Original Classifications** | Re-entrancy | Access Control | Arithmetic | Unchecked LLC | Total |
|---|---|---|---|---|---|
| Re-entrancy | | | 16 | | 16 |
| Access Control | | | 1 | | 1 |
| Arithmetic | | | | | |
| Unchecked LLC | 1 | | 6 | | 7 |
| Total | 1 | | 23 | | 24 |

## 6.2 Experiment Osiris

Table 6, 7 and 8 show the raw data captured for each Static Analysis tool, in terms of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) when testing the tool's effectiveness at detecting vulnerabilities within the smart contract tested. Table 6, 7, and 8 show the smart contracts as per their vulnerability classification under Column "Vulnerabilities" and the detection rate under Rows labelled "TP","TN", "FP" and "FN".

**Table 6: Dataset tested by Osiris**

| **Vulnerabilities** | **Read** | **TP** | **TN** | **FP** | **FN** |
|---|---|---|---|---|---|
| **SmartBugs Dataset** | | | | | |
| Re-entrancy | 18/31 | 18 | 0 | 0 | 1 |
| Access Control | 4/18 | 0 | 3 | 2 | 0 |
| Arithmetic | 10/15 | 32 | 0 | 1 | 1 |
| Unchecked LLC | 18/26 | 0 | 5 | 12 | 0 |
| Total | 50 | 59 | 8 | 15 | 2 |
| **SWC Dataset** | | | | | |
| Total | 19 | 20 | 5 | 11 | 4 |
| **Datasets Total** | 69 | 79 | 13 | 26 | 6 |

## 6.3 Experiment Oyente

**Table 7: Dataset tested by Oyente**

| **Vulnerabilities** | **Read** | **TP** | **TN** | **FP** | **FN** |
|---|---|---|---|---|---|
| **SmartBugs Dataset** | | | | | |
| Re-entrancy | 18/31 | 18 | 0 | 0 | 1 |
| Access Control | 4/18 | 0 | 4 | 2 | |
| Arithmetic | 10/15 | 32 | 0 | 1 | 1 |
| Unchecked LLC | 20/26 | 0 | 4 | 16 | 0 |
| Total | 48 | 50 | 8 | 19 | 2 |
| **SWC Dataset** | | | | | |
| Total | 20/117 | 5 | 11 | 0 | 4 |
| **Datasets Total** | 72 | 31 | 19 | 19 | 6 |

## 6.4 Experiment Slither

**Table 8: Datasets tested by Slither**

| **Vulnerabilities** | **Read** | **TP** | **TN** | **FP** | **FN** |
|---|---|---|---|---|---|
| **SmartBugs Dataset** | | | | | |
| Re-entrancy | 29/31 | 28 | 0 | 0 | 1 |
| Access Control | 18/18 | 14 | 0 | 0 | 3 |
| Arithmetic | 15/15 | 0 | 0 | 5* | 0 |
| Unchecked LLC | 26/26 | 26 | 0 | 0 | 3 |
| Total | 88 | 68 | 0 | 0 | 7 |
| | | | | | |
| **SWC Dataset** | | | | | |
| Total | 97 | 47 | 12 | 7 | 34 |

| | | | | | |
|---|---|---|---|---|---|
| **Datasets Total** | 185 | 111 | 12 | 7 | 41 |

**Revert 5 vulnerabilities as follows = 2 to SWC 100 and 3 to SWC 131*

## 6.5 Results

Table 9 shows the confusion matrix for each of the Static Analysis tools and the vulnerabilities the tools attempted to detect. The data captured from Tables 6, 7 and 8 was utilised to generate the confusion matrix in Tables 9 and 10. Table 10 shows the combined result for each Static Analysis tool for the entire vulnerability dataset employed.

**Table 9: Vulnerability Type against Static Analysis Tool**

| **Static Analysis Tool** / **Vulnerability** | **Osiris** | **Oyente** | **Slither** |
|---|---|---|---|
| **Re-entrancy** | | | |
| Recall | 0.947368 | 0.947368 | 0.965517 |
| Precision | 1 | 1 | 1 |
| Accuracy | 0.947368 | 0.947368 | 0.965517 |
| F1-Score | 0.972973 | 0.972973 | 0.982456 |
| **Access Control** | | | |
| Recall | 0 | 0 | 0.823529 |
| Precision | 0 | 0 | 1 |
| Accuracy | 0 | 0 | 0.823529 |
| F1-Score | 0 | 0 | 0.903226 |
| **Arithmetic** | | | |
| Recall | 1 | 0.969697 | 0 |
| Precision | 0.969697 | 0.969697 | 0 |
| Accuracy | 0.941176 | 0.941176 | 0 |
| F1-Score | 0.984629 | 0.969697 | 0 |
| **Unchecked LLC** | | | |
| Recall | 0 | 0 | 0.896552 |
| Precision | 0 | 0 | 1 |
| Accuracy | 0 | 0 | 0.896552 |
| F1-Score | 0 | 0 | 0.945455 |
| **SWC** | | | |
| Recall | 0.833333 | 0.555555 | 0.580246 |
| Precision | 0.645161 | 1 | 0.870370 |
| Accuracy | 0.625000 | 0.800000 | 0.590000 |
| F1-Score | 0.727272 | 0.714285 | 0.672000 |

**Table 10: Overall Combined Datasets**

| | Recall | Precision | Accuracy | F1-Score |
|---|---|---|---|---|
| **Osiris** | 0.929412 | 0.752381 | 0.741935 | 0.831579 |
| **Oyente** | 0.873016 | 0.591399 | 0.643411 | 0.705129 |
| **Slither** | 0.735100 | 0.940678 | 0.719298 | 0.825279 |

## 6.6 Discussion

The experiment was limited by the number of tools tested and the level of understanding of vulnerabilities within Solidity. Research was hampered by the lack of coordination within the community to categorise vulnerabilities, an opinion shared by other researchers (e.g., [23] and [5]), hence the introduction of reclassification in this experiment. SmartBugs and SWC Registry do provide a valuable service in providing documented and categorised datasets. Whilst writing this report, SWC added SWC 136 to the registry demonstrating the commitment to maintaining a registry of vulnerabilities.

Using the confusion matrix, a Ture Positive and Ture Negative are desired. However, this experiment tested for vulnerabilities, therefore, the consequence of a False Negative i.e., undetected vulnerability is more severe than a False Positive i.e., wrongly detected vulnerability. An undetected vulnerability will be a security risk to smart contracts deployment. A False Positive may result in increased production time to find a nonexciting vulnerability. Precision values measure the influence of the False Positive, and a type I error, while Recall measures the influence of False Negative and is a type II error.

An imbalance in the dataset existed in the experiment, as some tools read more contracts than others and some tools were not designed to detect certain vulnerabilities. F1 Score metric is an appropriate metric to handle this imbalanced data. The F1 -Score is the weight average of Recall and Precision, i.e., the metric harmonises the mean of Recall and Precision. The closer to the value 1 an F1-Score is, the closer the model is in predicting the classification of an outcome.

### 6.6.1 Discussion Individual Vulnerabilities

Examining for Re-entrancy vulnerability, Slither recorded the highest prediction (Table 9) with a low level of False Negatives as indicated by the highest Recall value in Table 9. Both Osiris and Oyente produced the same a high prediction score (Table 9) indicating all Static Analysis tools tested are be suitable for Re-entrancy testing, agreeing with [32] who recommended Slither and Oyente in their research. It is noted [31] did not record re-entrancy detection for Osiris. While [27] found Slither detected all occurrence of re-entrancy, it had a high rate of False Positives, which contradicts this report's findings. The Oyente results (Table 7) in this experiment contradicted [1] where the authors stated Oyente they returned false warnings for re-entrancy from "…problematic smart contracts".

Access Control was only detected by Slither with a strong prediction score F1 Score in Table 9. While the Recall value indicates a number of False Negatives. These findings echo [32] stating "There was no effective countermeasure for detecting the 'Access Control'...".

Concerning Arithmetic vulnerability (SmartBugs dataset) detection, Osiris and Oyente performed well with Osiris returning a stronger prediction as observed in Table 9. This experiment returned low False Positives (Table 7) for Oyente which contradicts two previous studies [27, 32] in which a high degree of False Positive was found. An explanation for this divergence was not established, noting different datasets utilised. [31] did not record detection for Arithmetic vulnerabilities for Oyente which contradicts [27] and [32].

Slither was not designed to detect Arithmetic vulnerabilities such as Over/Under Flow Integers, however, Slither returned 5 detections for a different vulnerability type on testing. For example, Slither detected an issue in the smart contract "overflow_single_tx.sol" on lines 36, 42 and 48 recording it as "unused local variable" a SWC 131 vulnerability, which can cause "increase in computation and unnecessary gas consumption". The same code lines were labelled by SmartBugs as "overflow escapes to publicly-readable storage", an Arithmetic vulnerability. Of the 5 detections above, 2 were "Secret" SWC 100 vulnerabilities, and 3 were SWC 131 vulnerabilities, therefore the detections were recorded as SWC detections.

Unchecked LLC vulnerabilities were only detected by Slither with a high degree of predictability (Table 9), which were similar to previous findings [32]. Osiris and Oyente respectively show a high degree of False Positives for Unchecked LLC smart contracts (Tables 6 and 7). On examination, the False Positives were attributed to falsely detecting nonexciting vulnerabilities of another type and not Unchecked LLC.

Tests conducted on the SWC registry dataset, considered the tools detection design. As observed from Table 9, Osiris returned the highest predictive value (F1 Score) and Recall value (Table 9). Both Oyente and Slither returned low Recall values indicating a high False Negative, undetected vulnerabilities. Oyente's F1 Score was marginal higher than Slither, however, the low Recall gives rise to concerns using Oyente and Slither for the SCW registry vulnerabilities. Oyente and Osiris have a very strong correlation examining Smartbugs contracts, however, Oyente and Osiris diverge on the examination of the SWC registry contracts.

### 6.6.2 Discussion Combined Vulnerabilities

Observing the overall results, as presented in Table 10, Slither's Precision value exceeds Osiris and Oyente, with Oyente mostly likely to miss a detection with to lowest value for Recall. Table 10 illustrates Oyente to have less trustworthiness with a lower Accuracy value, combined with lower Precision values. This is reflected in the lower F1-Score as a predictor for Oyente ability to detect vulnerabilities. Overall, Osiris produced the higher F1-Score and will give a better prediction at vulnerability detection, followed closely by Slither and next by Oyente with the lowest prediction.

It is apparent from the experiment that categories vulnerability type in smart contracts are uncertain. Several contracts contained vulnerabilities not labelled and if not manually check would have skewed results. The sample size in this experiment was small to allow for manual inspection.

# 7 Conclusion and Future Work

The question *"How effective are different docker static analysis testing tools at detecting different vulnerabilities in Solidity generated Smart Contracts?"*

The objective of the experiment was to test the effectiveness of static analysis tools in docker at detecting smart contract vulnerabilities from a given dataset. There is no standardisation in labelling of vulnerabilities and the reliance on developers to categorise vulnerabilities is mishap. The experiment included testing the dataset and subsequently re-categorizing smart contracts with additional vulnerabilities which were not originally labelled. Works conducted included, running testing static analysis tools in docker, testing the smart contracts against the tools, recording the results, and interrupting the results via the confusion matrix.

The objectives were achieved, and the research question addressed concerning how effective the different static analysis tools were. The experiment produced meaningful results allowing comparison between the tools. Furthermore, as anticipated the dataset required adjustment, highlighting concerns regarding datasets used for testing. While the research question involved different vulnerabilities, the level of reclassification of vulnerability smart contract was noteworthy. This research has gone over some old ground, however, in past papers the dataset was given little attention and in many cases the vulnerabilities listed were not verified independently [5]. This research also demonstrated the issues concerning vulnerability labelled datasets.

Vulnerabilities were observed in Re-entrancy, Unchecked LLC and Access Control which can lead to the additional vulnerability of an arithmetic nature e.g., over/under flow integrity.

Slither was found to be the most effective and versatile tool having the ability to read more contracts and detect more vulnerability types. However, weakness in returning False Negatives were discovered. Oyente and Osiris produced similar result throughout the experiment, allowing for detection of unlabelled vulnerabilities. In this experiment Oyente performance was considered the least effective. While Slither and Osiris are effective tools, each has detection capabilities that the other does not, therefore a solution suggested going forward is to setup a framework incorporating Osiris and Slither to conduct tests giving greater scope in detection.

## 7.1 Future works

The static analysis tools did not scan all the same smart contracts and the number of contracts read varied, with Slither scanning the most. In future works, to mitigate against this weakness, the latest or a more recent version of the Solidity complier should be installed in the docker image. However, in this experiment, attempts to build a docker image with a more recent complier could not be achieved in the time available. Alternativity, increasing the dataset size in the experiment may increase the quantity of contracts scanned. Increasing the sample size will increase resources required to individual assess each contract for vulnerabilities and rechecking after testing.

This experiment while small demonstrated the capability Docker images Osiris, Oyente and Slither at testing and the different levels of effectiveness of each of the tools at testing. Futures work could consider utilising docker static analysis images for testing as part of a larger framework. Furthermore, consideration should be given to a standardised classification vulnerability dataset for testing and registry of vulnerabilities similar to the SEI CERT Oracle Coding Standard for Java, hosted by Carnegie Mellon University.

# References


[1] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, 'Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey'. arXiv, Sep. 16, 2020. Accessed: Mar. 05, 2022. [Online]. Available: http://arxiv.org/abs/1908.08605

[2] 'ISO/TR 23455:2019(en), Blockchain and distributed ledger technologies — Overview of and interactions between smart contracts in blockchain and distributed ledger technology systems'. https://www.iso.org/obp/ui/#iso:std:iso:tr:23455:ed-1:v1:en (accessed Jun. 04, 2022).

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, 'A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses', *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, May 2021, doi: 10.1145/3391195.

[4] A. Bouichou, S. Mezroui, and A. E. Oualkadi, 'An overview of Ethereum and Solidity vulnerabilities', in *2020 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, Nov. 2020, pp. 1–7. doi: 10.1109/ISAECT50560.2020.9523638.

[5] M. Ren *et al.*, 'Empirical evaluation of smart contract testing: what is the best choice?', in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, New York, NY, USA, Jul. 2021, pp. 566–579. doi: 10.1145/3460319.3464837.

[6] 'SWC-100 · Overview'. http://swcregistry.io/ (accessed Jul. 10, 2022).

[7] 'smartbugs/dataset at master · smartbugs/smartbugs · GitHub'. https://github.com/smartbugs/smartbugs/tree/master/dataset (accessed Aug. 05, 2022).

[8] A. Panarello, N. Tapas, G. Merlino, F. Longo, and A. Puliafito, 'Blockchain and IoT Integration: A Systematic Survey', *Sensors*, vol. 18, no. 8, Art. no. 8, Aug. 2018, doi: 10.3390/s18082575.

[9] S. Solidity Team, 'Solidity Programming Language', *Solidity Programming Language*. https://soliditylang.org// (accessed Aug. 04, 2022).

[10] 'Ethereum', *ethereum.org*. https://ethereum.org (accessed Aug. 04, 2022).

[11] 'The HUGE growth of the solidity developer salary in 2022', *Plexus Resource Solutions*, Mar. 08, 2022. https://www.plexusrs.com/growth-solidity-developer-salary/ (accessed Jun. 04, 2022).

[12] T. Sharma, Z. Zhou, A. Miller, and Y. Wang, 'Exploring Security Practices of Smart Contract Developers'. arXiv, Apr. 24, 2022. Accessed: May 13, 2022. [Online]. Available: http://arxiv.org/abs/2204.11193

[13] N. Atzei, M. Bartoletti, and T. Cimoli, 'A survey of attacks on Ethereum smart contracts', 1007, 2016. Accessed: Mar. 05, 2022. [Online]. Available: http://eprint.iacr.org/2016/1007

[14] E. Banisadr, 'How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight', *Medium*, Feb. 03, 2019. https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530 (accessed Jun. 04, 2022).

[15] B. Mueller, ‘Smashing Ethereum Smart Contracts for Fun and Real Profit’, *HITB SECCONF Amsterdam and ConsenSys Diligence*, p. 54, 2018.
[16] S. User, ‘Blockchain Attack Vectors: Vulnerabilities of the Most Secure Technology’, *Apriorit*. https://www.apriorit.com/dev-blog/578-blockchain-attack-vectors (accessed May 06, 2022).
[17] ‘Thoughts on The DAO Hack’, *Hacking Distributed*. https://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/ (accessed Jun. 04, 2022).
[18] vasa, ‘HackPedia: 16 Solidity Hacks/Vulnerabilities, their Fixes and Real World Examples’, *HackerNoon.com*, Aug. 15, 2020. https://medium.com/hackernoon/hackpedia-16-solidity-hacks-vulnerabilities-their-fixes-and-real-world-examples-f3210eba5148 (accessed May 08, 2022).
[19] N. F. Samreen and M. H. Alalfi, ‘A Survey of Security Vulnerabilities in Ethereum Smart Contracts’, *arXiv:2105.06974 [cs]*, May 2021, Accessed: Apr. 22, 2022. [Online]. Available: http://arxiv.org/abs/2105.06974
[20] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, ‘TXSPECTOR: Uncovering Attacks in Ethereum from Transactions’, p. 19, Aug. 2020.
[21] C. F. Torres, J. Schütte, and R. State, ‘Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts’, in *Proceedings of the 34th Annual Computer Security Applications Conference*, San Juan PR USA, Dec. 2018, pp. 664–676. doi: 10.1145/3274694.3274737.
[22] J. Feist, G. Grieco, and A. Groce, ‘Slither: A Static Analysis Framework For Smart Contracts’, in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019, pp. 8–15. doi: 10.1109/WETSEB.2019.00008.
[23] ETSI, ‘GS PDL 011 Permissioned Distributed Ledger (PDL); Specification of Requirements for Smart Contracts’ Architecture and Security’. Accessed: Jun. 05, 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/PDL/001_099/011/01.01.01_60/gs_PDL011v010101p.pdf
[24] M. Saad *et al.*, ‘Exploring the Attack Surface of Blockchain: A Comprehensive Survey’, *IEEE Communications Surveys Tutorials*, vol. 22, no. 3, pp. 1977–2008, 2020, doi: 10.1109/COMST.2020.2975999.
[25] A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, ‘A Mechanism to Detect and Prevent Ethereum Blockchain Smart Contract Reentrancy Attacks’, *Frontiers in Computer Science*, vol. 3, 2021, Accessed: Apr. 20, 2022. [Online]. Available: https://www.frontiersin.org/article/10.3389/fcomp.2021.598780
[26] D. Perez and B. Livshits, ‘Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited’, 2021, pp. 1325–1341. Accessed: May 01, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/perez
[27] A. Ghaleb and K. Pattabiraman, ‘How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection’, in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA, Jul. 2020, pp. 415–427. doi: 10.1145/3395363.3397385.
[28] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, ‘SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses’, in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 227–239. doi: 10.1109/ASE51524.2021.9678888.
[29] C. Benabbou and Ö. Gürcan, ‘A Survey of Verification, Validation and Testing Solutions for Smart Contracts’, in *2021 Third International Conference on Blockchain*

*Computing and Applications (BCCA)*, Nov. 2021, pp. 57–64. doi: 10.1109/BCCA53669.2021.9657040.
[30] T. Petar, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, 'Securify: Practical Security Analysis of Smart Contracts', in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, Oct. 2018, pp. 67–82. doi: 10.1145/3243734.3243780.
[31] M. di Angelo and G. Salzer, 'A Survey of Tools for Analyzing Ethereum Smart Contracts', in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, Apr. 2019, pp. 69–78. doi: 10.1109/DAPPCON.2019.00018.
[32] S. Ji, D. Kim, and H. Im, 'Evaluating Countermeasures for Verifying the Integrity of Ethereum Smart Contract Applications', *IEEE Access*, vol. 9, pp. 90029–90042, 2021, doi: 10.1109/ACCESS.2021.3091317.
[33] Z. A. Khan and A. S. Namin, 'A Survey on Vulnerabilities of Ethereum Smart Contracts'. arXiv, Dec. 28, 2020. Accessed: Jul. 21, 2022. [Online]. Available: http://arxiv.org/abs/2012.14481
[34] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, 'SmartCheck: Static Analysis of Ethereum Smart Contracts', in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2018, pp. 9–16.
[35] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, 'Making Smart Contracts Smarter', in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna Austria, Oct. 2016, pp. 254–269. doi: 10.1145/2976749.2978309.
[36] S. Akca, A. Rajan, and C. Peng, 'SolAnalyser: A Framework for Analysing and Testing Smart Contracts', in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 482–489. doi: 10.1109/APSEC48747.2019.00071.
[37] 'Docker Hub Container Image Library | App Containerization'. https://hub.docker.com/ (accessed Aug. 09, 2022).