THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

School of Information Technology and Electrical Engineering,

University of Queensland.

Static Analysis of Solidity Smart Contracts

By Nikhil Naik

Submitted for the degree of Bachelor of Engineering (Honours) in the division of Software Engineering

13 June 2022

Nikhil Naik
n.naik@uqconnect.edu.au
June 13, 2022

The Head of School of
Electrical Engineering Faculty
of Engineering, Architecture and
Information Technology
The University of Queensland
St Lucia QLD 4072

Dear Professor Michael Bruenig,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the division of Computer Systems Engineering, I present the following thesis entitled "*Static Analysis of Solidity Smart Contracts*". This work was performed under the supervision of Dr Naipeng Dong.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Nikhil Naik

Acknowledgements

I would like to acknowledge and thank Dr Naipeng Dong, for her continual and committed guidance and support in completing this thesis project. Dr Naipeng Dong assisted at the beginning of this project immensely in pointing this project scope towards a direction which was achievable and demonstrated significant contribution towards this field of Program Analysis. During the project she provided continual constructive feedback on the project's methodology of producing Control Flow Diagram logic, the proposed Solidity Static Analysis Tool '*PySolSweep*' as well as the comparative experiment to evaluate the projects tool against existing Solidity Static Analysis tools. Dr Naipeng Dong's critical input on assessment items and Analysis of results contributed significantly to producing this thesis project.

Abstract

Static Analysis is a Program Analysis method applied to programming language code, to detect bugs, vulnerabilities and potential countermeasures within code. This Thesis Project focused on Static Analysis of Ethereum Smart Contracts, written in the programming language Solidity. The aim was to construct a Solidity Static Analysis tool, which overcame the gaps and limitations of current tools. This project investigated three major Bug Attack Themes (BAT's) of Overflow/Underflow, Syntax and DAO. Each BAT corresponded to a set of bugs, vulnerabilities and countermeasure implications in a Smart Contract, from Solidity documentation and academic papers reviewed. The limitations of current Solidity Static Analysis Tools illuminated a lack of BAT coverage, missing logic for new bugs/vulnerabilities, poor compatibility/usability and a lack of a solution for that bug or vulnerability.

The project produced 35 Control Flow Diagrams (CFD's) which extended on the logic of existing bugs/vulnerabilities, as well as provided logical violation paths for new bugs, vulnerabilities and countermeasures within the three BAT's. The CFD logic was implemented within the proposed project Solidity Static Analysis tool '*PySolSweep*', which overcame existing limitations to integrate increased BAT coverage, improved compatibility/usability and solutions for bugs, vulnerabilities or countermeasures detected in a Smart Contract. A comparative evaluation experiment conducted with '*PySolSweep*' against existing Solidity Static Analysis tools, reinforced the project's approach of increased BAT coverage detecting more bugs, vulnerabilities and countermeasures. With current tools approach of random bugs detected, lacking significant protection against the three BAT's. The experiment also illuminated Blockchain deployed Smart Contracts containing large volume of medium and high impact bugs, vulnerabilities and countermeasures within the scope of Overflow/Underflow, Syntax and DAO

This Thesis Project demonstrated a successful approach towards improving on the limitations of current Solidity Static Analysis tools lacking coverage against specific BAT attacks. With future work of implementing the tool using a Tree Parser infrastructure. And a more in-depth evaluation with larger Smart Contract dataset and comparative tools further reinforces and validates this project's contributions to the area of Static Program Analysis.

Contents

List of Figures

List of Tables

1.0 Introduction

The main area of focus for this project incorporates that of evaluating and developing Static Program Analysis methods to be used in detecting vulnerabilities and bugs within Ethereum deployed Smart Contracts. Program Analysis consists of either being Static or Dynamic in implementation [1]. Static Analysis involves finding bugs and vulnerabilities within source code, prior to runtime execution. This type of Whitebox testing works by comparing source code to a set of predefined rules in an automated process [2]. Whilst Dynamic Analysis follows a Blackbox visibility implementation, by finding bugs and vulnerabilities at time of runtime execution [3]. Both methods of Program Analysis play an integral role in achieving software security requirements of confidentiality, integrity, authenticity, availability and non-repudiation [4].

Smart Contracts are placed on the Ethereum Blockchain, which is a decentralised blockchain based platform, which allows for Smart Contracts to be built and run without any downtime, fraud, or interference from any third-party client. It also allows for applications to run in a decentralised architecture, such that no single entity controls or sources information, the data is shared in the form of a peer-to-peer network [5].

Within the Ethereum Blockchain, code is executed via the EVM (Ethereum Virtual Machine) on the block chain with a Smart Contract. Smart Contracts are decentralised, and hence work with reading and writing data to the blockchain. The blockchain can be visualised as a database which confirms whether transactions have taken place [6]. The Smart Contract fits into this by executing predefined conditions that are met, only then can the transfer occur in the blockchain. The language used to write Smart Contracts includes that of Solidity. This language is object oriented and allows for writing Contracts on the Ethereum platform. Within a Solidity Contract, there is a collection of code that contains functions and the data's state, which resides at an address on the Ethereum blockchain [7].

The implementation of Smart Contracts on the blockchain is highly money driven, with the Ethereum market being worth over $520 billion [8]. Given so, the financial implications of a security vulnerability or bug being exploited in a Smart Contract could be hugely detrimental. Some common vulnerabilities and bugs are categorised into the following areas [9]:

- Overflow/Underflow Bugs
- Syntax Bugs
- DAO Bugs
- Parity Sig Multi Wallet Bugs

The Overflow/Underflow bugs relate to an adversary exploiting numerical ranges as well as arithmetic operations within code that either feeds a function erroneous, malicious, or out of scope data values. Syntax bugs involve errors or vulnerabilities in code which abstain or are vulnerable from the confines of the rules and format of a programming language [19]. The DAO (Decentralised Autonomous Organisation) bugs related to functions in code which interact with a transfer or withdrawal of funds (Eth). Finally, the Parity Sig Multi Wallet bugs incorporated a Smart Contracts constructor initialization logic from external libraries as well as suicide or kill mechanisms implementation [20].

Many solutions of Static and Dynamic program Analysis software tools are available such as SmartCheck, Slither, Mythril and Securify. However, these tools have their many limitations and gaps such as having many dependencies to build the tool. As well as targeting a broad range of bugs and vulnerabilities, these tools fail to detect every bug or vulnerability within a certain category from Overflow/Underflow, Syntax, DAO and Parity Sig Multi Wallet issues. The extent towards this limitation will later be discussed in the background information [10].

1.1 Aims

The main aim of this project is to design and develop a Static Program Analysis tool '*PySolSweep*', that can detect and cover a specific area of bugs and vulnerabilities within the Solidity code of Smart Contracts. Implementing methods derived from constructing Control Flow Diagrams, which represent the logical flow of steps towards bug/vulnerability detection.

1.2 Scope

1.2.1 In Scope

The following objectives will be produced as part of this Thesis project:

- Control Flow Diagrams for each bug detected
- Python Static Analysis tool which offers coverage for:
    - Overflow Underflow Attack
    - Syntax Attack
    - DAO Attack
- Benchmark Criteria for collated deployed Smart Contract code dataset
- Analysis of PySolSweep and existing Static Analysis tools on deployed Smart Contracts
- Analysis of Attack Theme bug detection approach to Static Analysis
- Recommendations of how Solidity Static Analysis tools can be improved to defend against a future attack

By developing this solution in the Python programming language, complex dependencies and build times to Statically analyse Smart Contracts will be reduced. As well as by specifying the categorical areas of bug and vulnerability detection that the issue of ambiguity around what bugs are detected in current software solutions can be mitigated, which will furthermore be discussed in Analysis of existing academic papers.

1.2.2 Attack Types

To ensure the scope is kept sizable and achievable, the Parity Sig Multi-Wallet attack will not be covered, since this attack requires an amalgamation between Static and Dynamic program

Analysis. This project focuses purely on the Static program Analysis of Solidity Smart Contracts. Hence the attack themes detection methods that will be implemented and investigated incorporate that of Overflow/Underflow, Syntax and DAO bugs. These attack themes contain bugs specific to them as summarised in **Table 1**.

Table 1: Attack Themes Bug Breakdown

Attack Theme	Number Of Related Bugs
Overflow/Underflow	*6*
Syntax	*24*
DAO	*5*

1.2.3 Dataset

To evaluate the success of this tool, an experiment will be conducted on currently deployed Smart Contracts comparing the project's Static Analysis tool to existing tools. The minimum goal is to better these tools in detection of a set of specified bugs and vulnerabilities from the categories of Overflow/Underflow, Syntax, DAO and Parity Sig Multi Wallet bugs. As well as match the detection accuracy of existing Static Analysis tools available. The scope of the project focuses on Solidity Smart Contracts deployed on 'Etherscan', which allows for Smart Contract code to be viewed of deployed Smart Contracts and their transactions.

1.2.4 Innovation

The scope of innovation of this project, incorporating the developed Python script Static Analysis tool '*PySolSweep'* will produce a detailed bug report when performing Analysis on a Solidity Smart Contract. This automatically generated report will contain the type of vulnerability, location of vulnerability line in code, risk of vulnerability, category of vulnerability, potential attacks from vulnerability, explanation of bugs/vulnerabilities in general terms, potential solution to vulnerability in code and a score that corresponds to an overall safety rating of the Smart Contract. The motivation for this was that stakeholders can use this score to determine whether they deploy their Smart Contract prematurely, as bugs/vulnerabilities detected later have extensive Contracts to modify and re-deploy the Smart Contract.

1.3 Relevance

This project is driven by the current gaps and limitations of Static Analysis tools for Smart Contract vulnerability and bug detection as well as the rapidly evolving environment of Smart Contracts. With the ever evolving and changing nature of new exploitation methods, regular updates to the Solidity language opens the door for attacks. Such attacks due to bug and vulnerability exploits include that of the DAO attack in 2016 in which $60 million worth of Ether was stolen. As well as the parity wallet hack in 2017 that had $30 million Ether stolen [11]. Both attacks were the direct result of vulnerabilities in the Solidity source code, something that Static Analysis tools could have detected. Even a more recent attack that occurred on August 13th 2021, in which crypto currency platform Poly Network was hacked, with a theft of $600 million [12]. These attacks further emphasise both the investigation and further development of Static Analysis tools to highlight and prevent vulnerabilities from being exploited.

The first Solidity Smart Contract was deployed in June 2015, making the programming language as of 2022 just 7 years in operation [21]. Other programming languages such as Python, C++, C and Java have existed for multiple decades. This disparity in first conception is cause for concern for Solidity, as the programming language is still being updated as well as having much less datasets on vulnerabilities and bugs that can be exploited due to being so new in release.

With both the continual evolution of Smart Contract attacks and shortcomings of current Solidity Static Analysis tools, these reasons support the goals and purpose of this thesis project to design and develop an improved Static Program Analysis tool. As well as reinforce key findings made in the research area of Static Analysis of Solidity Smart Contracts.

2.0 Background

2.1 Static Analysis of Integer Overflow of Smart Contracts in Ethereum [13]

This study explored 11 variations in integer overflow vulnerability in Solidity as well as proposing a Static Analysis tool to detect common integer overflow vulnerabilities in Solidity Smart Contracts. An overarching sentiment surrounding detection tools was that the Integer overflow vulnerabilities cannot all be detected by most tools, leading to exploits for attackers. The type of integer overflow was found to be divided into 3 variations of addition, subtraction and multiplication overflow/underflow. These individual vulnerabilities for each variation are summarised below:

Multiplication:

- **Feature 1:** Smart Contract source code not checking integer parameter passed in allows for an operation within range. Attacker can pass in large value into
  eg uint amount = len * user_val, to exceed the range
- **Feature 2:** Source code has statement that integer value is multiplied with another integer value which the result is <= to the integer state balance of the Contract
- **Feature 3:** Source code has statement that an integer variable multiply equal (*=) to the integer parameter of the mapping type eg amount *= _value

Addition:

- **Feature 1:** Source code has statement that variable equal to the integer parameter + another integer variable eg amount = p + _value
- **Feature 2:** Source code has statement that integer of mapping type = itself + integer parameter eg balances[Target] = balances[Target] + newToken
- **Feature 3:** Source code has statement that the integer variable of mapping type (+=) to the integer parameter eg allowed[Target][msg.sender] += newToken
- **Feature 4:** Source code has statement msg.sender + integer variable is <= the balance of Contract

Subtraction:

- **Feature 1:** 3 Overflow same as Features 1,2 and 3 in Addition replacing '+' with '-'.
- **Feature 2:** Loop body of code contains integer/state variable of mapping type (-=) to the integer

The paper illustrates the approach taken in the proposed tool following a logical multi-step process. This included existing tools using tree parse ANTLR (Another Tool for Language Recognition), which is a grammar Analysis generator based on lexical Analysis. This output then generates an XML AST (Abstract Syntax Tree), which represents the source code of a computer program that conveys the structure of the source code. Each node in the tree represents a construct occurring in the source code. The final step for detection involved running the tool XPath to evaluate bug or security violations in the XML AST. This existing approach for detection is summarised in **Figure 1**.

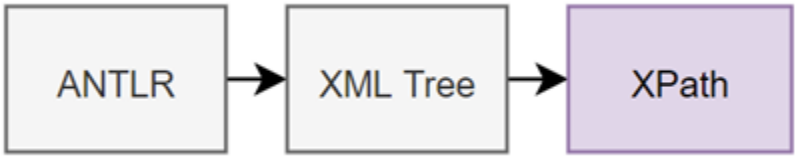


Figure 1: Existing Tools Static Analysis Approach

The key takeaways for design and general knowledge are that once one of the integer variables can be manipulated for example '*msg.value*' with no operations to check if the executions are within range is a common vulnerability for all three variations of integer multiplication, addition and subtraction underflow/overflow. The paper also found that on 7000 existing public Smart Contracts that 430 overflow/underflow bugs were detected, emphasising the further need to explore this area. The features listed above are constructive for the design process of a Static Analysis tool in what patterns to look for in Solidity Smart Contracts with regards to XML AST directing what occurs in the XPath tool for verification of bugs.

2.2 SmartCheck: Static Analysis of Ethereum Smart Contracts [14]

This study created and analysed the vulnerability detection tool SmartCheck, which converts Solidity code to XML based and runs it against XPath patterns to detect bugs and vulnerabilities. The paper was able to deduce the following security challenges with Solidity based Smart Contracts:

- **Varying Execution Environment:** Ethereum differs in centrally managed execution environments i.e. mobile, desktop or cloud
- **New Software Stack:** Ethereum Stack (EVM) still being developed with new vulnerabilities discovered
- **Limited ability to patch Contract:** Contract must be correct prior to deployment
- **Anonymous Financially motivated attacker:** High financial gain as well as anonymity difficult punishment of crime

These security challenges create motive and purpose to develop Static Analysis tools as well as increase the knowledge base of this area. Given so, the paper also divided the current bugs and vulnerabilities into 3 areas of syntax, functional, operational and developmental. Furthermore the potential pattern detection solution that its tool SmartCheck uses for overcoming and correcting these vulnerabilities:

- **Syntax Challenges**
    - **Balance Equality:** Avoid checking for balance equality since an attacker send funds to any account by mining eg
        - If this.balance == 10 eth (Very Vulnerable)
        - If this.balance >= 10 eth (Safe)
    - **Unchecked External Call:** Calls to external Contracts should fail, when sending eth check return value and handle errors eg
        - addr.send(10 eth) (Very Vulnerable)
        - If !addr.send(10 eth) revert (Vulnerable)
        - addr.transfer(10 eth) (Safe)
    - **DoS By External Contract:** The conditional (if/for/while) must not rely on external call; this call may fail causing incomplete execution
    - **Send vs Transfer:** Should use addr.transfer(x) which auto exception throw if transaction fail, so do not use send
    - **Re-Entrancy:**

```
Contract Fund {
    mapping ( address = > uint ) balances;
    function withdraw() public {
        if (msg. sender . call . value (balances[msg.sender ])())
            balances[msg. sender ] = 0;
        }
    }
```

Here msg.sender gets multiple refunds and gets all funds ether with recursive calls to withdraw before its share = 0.

With the corrective steps being

```
function withdraw() public {
    uint balance = balances[msg. sender ];
    balances[msg. sender ] = 0;
    msg.sender.transfer ( balance ) ;
    (state reverted ; balance restored if transfer fails)
}
```

The Pattern detects external function call followed by an internal function call

- **Malicious Libraries:** Pattern detects 3rd party libraries as external dependencies can lead to vulnerabilities
- **Using tx.origin:** Should use msg.sender instead of tx.origin for Authentication, for Contracts calling each other's functions

- **Functional Issues:**
  - **Integer Division:** Unsupported Floating Point/Decimals, Pattern detects "/" when numerator and denominator are both number literals
  - **Locked Money:** A Contract that received ether should contain withdraw (transfer/send/call), pattern checks if withdraw function but no withdraw operation
  - **Unchecked Math:** SafeMath library to check for numerical over/underflow
  - **Unsafe Type Inference:** Checks LHS is a variable and RHS is an integer, this could cause overflow
- **Operational Issues**
  - **Byte Array:** Use bytes instead of byte[] for less gas use, pattern detects byte[]
  - **Costly Loop:** If array length is too big function may exceed gas limit and never complete causing transaction to never be confirmed. Attacker can register many addresses causing DoS, looks for "for" statement with function call/identifier inside condition or looks for "while" statement with function call inside condition
- **Developmental Issues:**
  - **Token API Violation:** ERC20 API used for tokens which are transferable units/values in the Contract. Pattern detects a Contract inherited from a Contract

with token may throw exceptions from inside of one of the functions (approve/transfer/transferFrom)

- **Compiler Version Not Fixed:** ^0.4.19 (Vulnerable), 0.4.19 (Safe), pattern detects version operator "^"
- **Implicit Visibility Level:** Default viability is public, pattern detects function/variable declarations without visibility modifier

This paper also introduced the concept of using parse trees (**Figure 2**) prior to developing a pattern for each area. This would be useful in coverage of all areas for Static Analysis in the project, as well as understand the different logical pathways an attacker could take to exploit a bug/vulnerability.

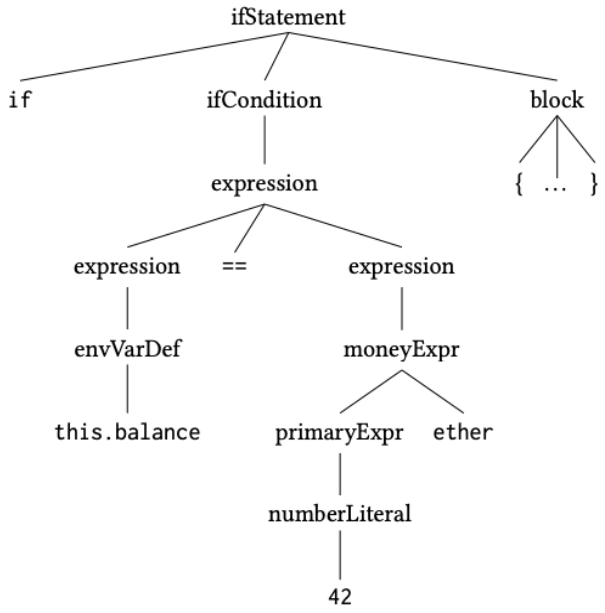


Figure 2: Bug Parse Tree for Balance Equality code [14]

Furthermore, by illustrating both good and bad examples of code, this is an important design takeaway, to aid in developing the Python source code for Static Analysis in this thesis project. By looking for those flaws, this could hopefully find bugs and vulnerabilities that other commercial Static Analysis tools overlooked.

2.3 Security Analysis Methods on Ethereum Smart Contract [15]

This paper explored and analysed three main attacks that occurred as well as technical reasons that caused these attacks. Which include the DAO attack, Parity Multi-Sig Wallet attack and integer underflow/overflow attack. The attack technical exploitation as well as solution is summarised below:

- **DAO Attack**
    - Re-Entrancy vulnerability
    - Repeatedly execute recursive calls for receiving and requesting funds with the DAO.sol Contract
    - Attacker kept withdrawing eth by requesting DAO Contract before updating balance of Smart Contract
    - Specifically the "*withdraw*" function was recursively called until Contract balance = 0
    - 2 Contracts with attacker Contract DAOAttacker.sol, sends ether to DAO.sol, attacker balance is updated, attacker requests to withdraw, fund sent back to attackers' Contract, fallback function for continuous withdrawal called
    - **Solution:** Use send()/transfer() instead of call.value()
- **Parity Multi-Sig Wallet Attack**
    - parity multisig wallets are Smart Contract programs which are used to manage digital assets by the wallet users
    - main problem caused by this attack is that all the public functions, such as initDayLimit and initMulitowned, in the WalletLibrary.sol Contract can be called by anyone without authorization
    - **Solution:** Use the internal modifier for functions instead of public and explicitly define library functions for the external invocations
- **Integer Over/Under Flow Attack**
    - Attacker sends eth to target Contract, target sends funds to attacker Contract, attackers fallback function triggered, and withdrawal requested again, Contract updates balance by subtracting 1 from 0, balance becomes -1
    - A Solidity compiler does not trigger any error flag to resolve the code with integer overflow/underflow problems.
    - **Solution:** Check if the integer stays in its byte range before any send operations, overflow/underflow problem can also be mitigated through using the arithmetic functions in the Solidity maths library named SafeMath.sol

In understanding these attacks, the vulnerabilities and bugs that exists in the Solidity code for all three are illuminated below:

- **Reentracy Problem:** The Solidity Smart Contract has an unnamed function called fallback function that does not have any arguments nor return values. Any code inside the fallback function would be executed until it finishes the remaining gas amount.
- **Exception Handling:** Failing to check the return values after a function call. A malicious user can invoke a caller Contract and cause its send function to fail purposefully.
- **Destroyable Contract:** Contract is susceptible to be destroyed by unauthorized users
- **Unchecked and failed send:** If there is no exception handling implemented at invoking send method, the balance would be updated as if it has been sent
- **Unsecured balance:** The Ether balance in a Contract is exposed because of the modifier *public* to theft by an anonymous caller of balance variable or constructor functions

These key vulnerabilities reinforce as well as increase the knowledge bank of existing vulnerabilities and bugs that exist in Solidity Smart Contracts. The paper also explored 2 methods for dynamic Analysis, which is not within the scope of this project however is important to understand that the same method of symbolic Analysis was used at runtime to identify patterns that could also be useful in Static Analysis:

- **MAIAN:** Trace vulnerabilities using the detection techniques across a long sequence of invocations of a Contract during its run-time. Finds violations in defined properties of traces in each of the Smart Contract executions
  - Symbolic Analysis: If vulnerable potential, this returns values for symbolic variables
  - Concrete Validation: Validates the results of Symbolic Analysis on a private fork checking if bugs at runtime are indeed detected
- **Graph Construction:** Graph Analysis used to detect abnormal Contract creation/behaviour
  - Anomaly Detection
  - 3 Types of Graphs:
    1. Money Flow Graph
    2. Contract Creation Graph
    3. Contract Invocation Graph
  - Steps of clustering, correlation, assortative in Graph Analysis

Finally, and important insight into the limitations and gaps that exist with current Static Analysis tools is evident below (**Figure 3**)

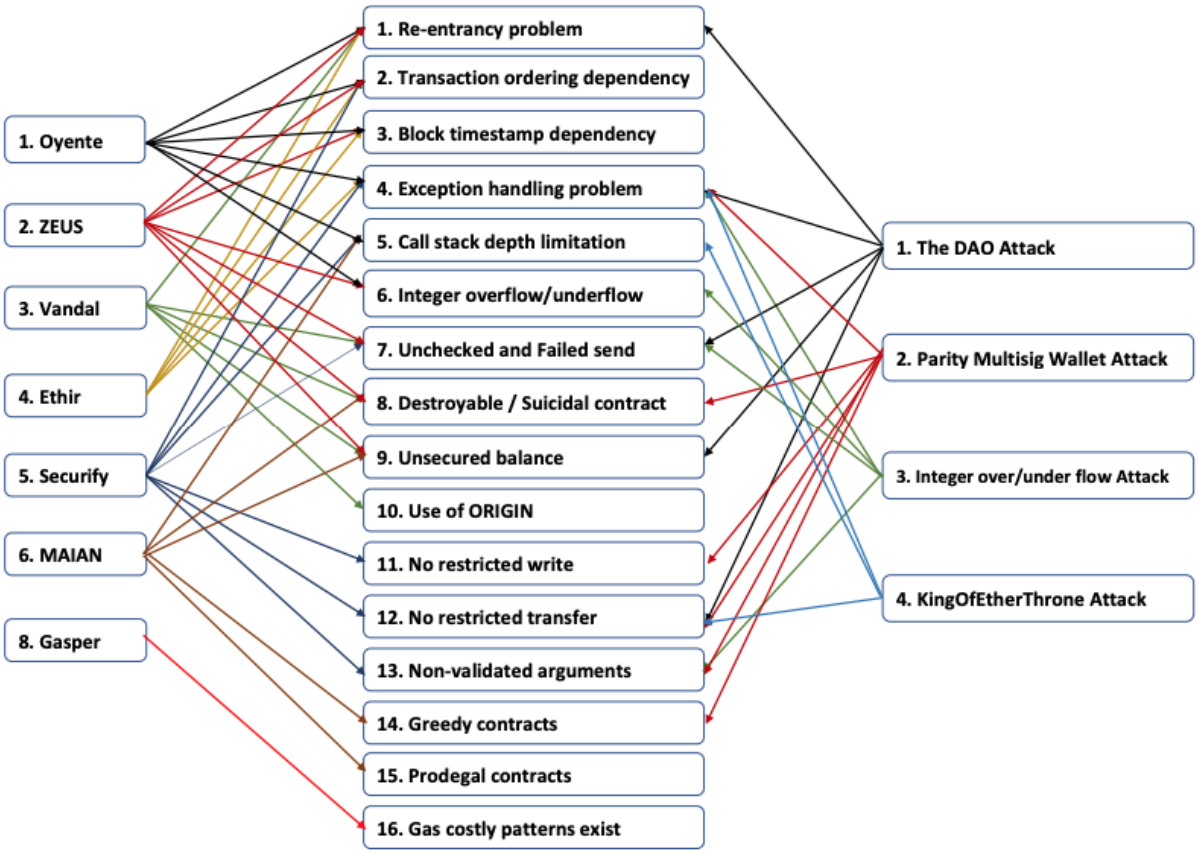


Figure 3: Summary of Attacks and their Vulnerability cause [15]

It is evident that no single Static detection software covers all potential attacks of Solidity Smart Contracts. Existing tools are capable of detecting various bugs and vulnerabilities, however, are not immune towards potential attacks. This limitation and gap in current commercial Static/Dynamic Analysis tools provides the motivations and purpose of this project. By reducing the ambiguity surrounding what specifically the Static Analysis tool detects, and which attack it prevents.

2.4 ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks [16]

This study explored a Dynamic Analysis tool ÆGIS, used for detecting vulnerabilities in Smart Contracts at runtime. This tool had the intention of detecting both the Reentrancy bug vulnerability as well as a new found bug vulnerability known as Access Control. Access Control as one exploit that was part of the Parity Mult-Sig attack, and can be summarised below:

- Incorrectly given user access control policies in Smart Contracts
- Parity Multi Sig: Developers forget a safety check that initWallet function, ensuring that the function can only be called once. As a result an attacker was able to gain ownership of the Contract by calling the initWallet function via the fallback function.

The paper also explained how the ÆGIS tool worked, by having a methodology split into a search for three attack patterns. This included control flow which looked for an Instruction that is control dependent on another instruction. The second pattern involved Data Flows, looking for Instruction is data dependent on another instruction. Finally was the Follow's attack pattern which identified when an instruction is executed after another instruction without having control or data dependent on the other instruction. Using this methodology solutions towards detecting both the Reentrancy and Access Control bug vulnerabilities.

- **Reentrancy Solution**
    - Attack pattern is true either if a control flow relation between 2 CALL instructions with the same destination OR two SSTORE instructions follow the previous control flow relation and both instructions write to the same location.
- **Parity Wallet Hack Solution**
    - Attack pattern is true either transaction with a control flow relation between a DELEGATECALL instruction and a CALLDATACOPY instruction, where the data of the CALLDATACOPY instruction flows into an SSTORE instruction

This insight provided a valuable understanding into proposal solutions towards the Reentrancy and access control vulnerability. With this information directly aiding development and knowledge base of this thesis project, since the aim was to not only detect but also provide potential solutions to overcoming bugs and vulnerabilities.

2.5 How Effective Are Smart Contract Analysis Tools [17]

This study proposed a Static Analysis tool evaluator Solidify, which analyses the accuracy of existing Smart Contract Static Analysis Tools using Bug Injection. With the major focus being on the false negative vulnerability bugs from these tools. The experiment was conducted by a method of Bug Injection into the Smart Contract source code. The Bugs were Code Snippets that caused a vulnerability to be detected by the Static Analysis security tool. By conducting this experiment, the study found a number of vulnerabilities that produced false negatives in these Static Analysis tools. These vulnerabilities and their potential solutions are summarised below:

- **Timestamp Dependency**
  - Block timestamp used to trigger some event
  - Malicious miners can change timestamp to benefit themself
  - GoverMental Ponzi scheme attack
  - function bug_tmstmp () public returns (bool) 2 {
    return block.timestamp >= 1546300;}
- **Unhandled Exceptions**
  - If exception thrown by the callee Contract, transaction is terminated, state reverted and false returned to caller Contract
  - unchecked returned values from caller Contract could be used to attack a Contract
  - King of the Ether Attack
  - function unhandledsend () public {
    callee.send (5 ether) ;}
  - Bad requires send to be checked for exceptions to make it secure
- **Integer Overflow/Underflow**
  - Exceeding range of integer value
  - function incrLockTime (uint _sec) public {
    lockTime[msg.sender] += _sec ;}
  - Bad Attacker can reset lockTime by calling incrLockTime and passing in value 256 which exceeds range 0 to 255
- **tx.origin**
  - This returns the first caller that originally sent the call
  - Purpose of Authentication
  - Issue of Phishing Attack
  - function bug_txorigin (address _recipient) public {
    require (tx.origin == owner) ;

_recipient . transfer (this.balance) ;}

- Bad Using tx.origin to withdraw funds
- Good use instead msg.sender == owner

- **Re-entrancy**
  - External calls of functions used by attackers to call a function within Smart Contract multiple times
  - DAO Attack
  - function bug_reEntrancy (uint256 _Amt) public {
    require (balances [msg.sender] >= _Amt) ;
    require (msg.sender.call.value (_Amt)) ;
    balances [msg.sender] -= _Amt ;}
  - Issue if call.value can be used by attacker to bug_reEntrancy() function multiple times and get funds multiple times
- **Unchecked Send**
  - Unauthorised eth transfer can be called by external users if visibility is public
  - function bug_unchkSend () payable public {
    msg.sender.transfer (1 ether) ;}
  - Issue if set as Public
- **Transaction Ordering Dependence (TOD)**
  - hanging the order of the transactions in a single block that has multiple calls to the Contract, results in changing the final output

The key takeaway from this study was that of the gaps that exist in current Static Analysis tools rigidity in Smart Contract vulnerability bug detection. Failure to detect such common bugs and vulnerabilities that formed previous attacks will be addressed in this thesis project tool.

2.6 Smart Contract: Attacks and Protections [18]

This paper introduced the idea of developers needing to employ strong security strategies prior to deployment to mitigate risk of vulnerability exploitation once deployed on the blockchain. By splitting attacks into 4 main attack rationale of Attacking Consensus Protocol, Smart Contract Bugs, OS Malware and Fraudulent Users. The paper used this rationale to investigate 7 common vulnerabilities as well as 10 common tools for Static Analysis. The attack types and risk severity are illuminated below:

- **Reentrancy** (High Severity and Risk); Potential to destroy Contract; Contains a revoke function
- **Overflow/Underflow** (Easy to Initiate requires user input);
- **Delegate Call** (High Severity and Risk) Allows anyone to do whatever they want with the Smart Contract state and msg.sender and msg.value do not change their values
- **Default Visibility** (Medium Severity and Risk) in Solidity is set to public, poor and sometimes ambiguous coding practice without explicitly stating that the function is private/public

By investigating 10 common Static Analysis tools with these bug vulnerabilities, that study deduced the following gaps and limitations:

    - Existing solutions fail to deal with more complex variations of vulnerabilities
    - Some solutions are limited to a small subset of vulnerabilities and not all or major ones
    - Smart Contracts never safe based on ever evolving Attack Vectors

Furthermore reinforcing the findings of previous studies, that existing Static Analysis tools have flaws in detecting common complex variations of vulnerabilities. Extending on this the paper also introduced a severity and risk rating system for bug vulnerabilities. As proposed in the goal and aim of this thesis project this concept of severity and risk rating provides valuable insights into how to categorise these vulnerabilities and a starting point to further label other bug vulnerabilities.

2.7 Security Evaluation and Improvement of Solidity Smart Contracts [22]

The final academic paper explored the implementation of Static Analysis for Smart Contracts with regards to existing Static Analysis tools' approach with their base set programming language. Focusing on the shortcomings and weaknesses of these Solidity Static Analysis tools of how and some vulnerabilities escape detection from these tools. To categorise vulnerabilities, the CWE (Common Weakness Enumeration) model was used to systematically represent vulnerabilities in the Solidity programming language. The key CWE classes and pillars include the following in **Figure 4**:

CWE-ID	CWE Description
CWE-20	*Improper input validation*
CWE-284	*Improper access control*
CWE-330	*Lack of randomness for random values*
CWE-345	*Insufficient Data Authentication verification*
CWE-400	*Uncontrolled resource consumption*
CWE-668	*Exposure to wrong external*
CWE-669	*Incorrect resource transfer*
CWE-682	*Incorrect calculation*
CWE-691	*Insufficient control flow management*
CWE-703	*Improper check/handling of exceptions*

Figure 4: Existing Tools Static Analysis Approach

The paper used these classes in an experiment to understand what the vulnerabilities of class and pillar detection could be with regards to CAPEC (Common Attack Pattern Enumeration and Classification) model. Using 400 Solidity Smart Contracts from Etherscan the following **benchmark criteria** for sourcing this dataset incorporated:

- Physical Lines of code
- Logical lines of code
- Minimum vulnerabilities detected
- Compliance with no compilation errors

- Representative to Solidity:
    - Exchange functions
    - Assembly usage
    - Low-level calls
    - Libraries
    - Structures
    - Arrays
- Vulnerabilities cover CWE
- Domain representative:
    - Gambling
    - Exchange
    - Games
    - Finance
    - Properties

This benchmark criteria were run, and existing tools performed Static Analysis on the dataset. The results (**Figure 5**) included Securify having the highest coverage for CWE-20, 703 and 284 with low accuracy. Whilst Slither and Remix had the highest detection for CWE-669, 691, 400 and 668 with high accuracy. Meanwhile Mythril and Osiris performed best with CWE-330 and 682. Overall no tool covered more than 66% of the CWE's, suggesting more room for detection and increased coverage of vulnerabilities.

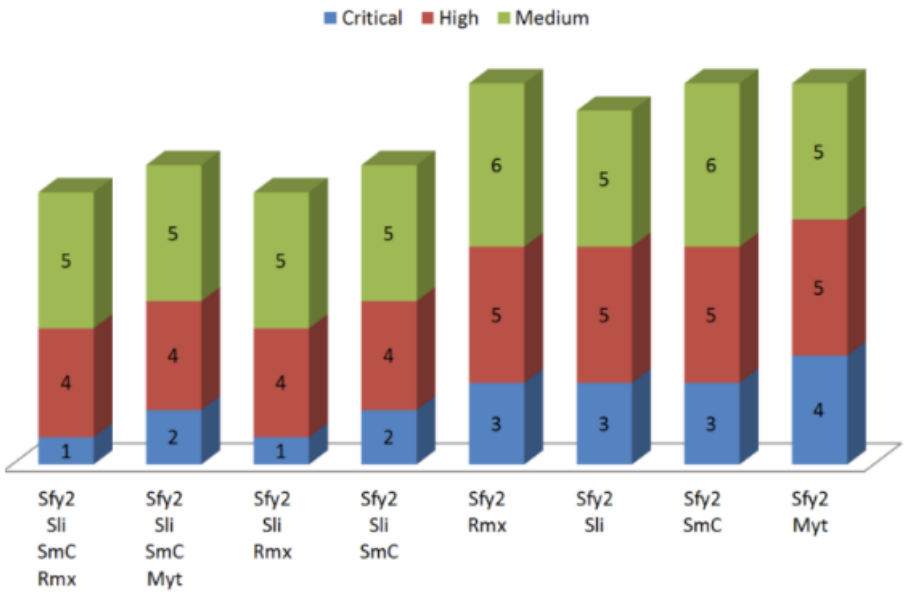


Figure 5: Existing Tools Bug Risk Distribution

With regards to the impact of this study on the project, the key takeaways of structuring and classifying vulnerabilities or bugs into a class similar to the CWE system is best practice for Analysis, since attacks derive based on a category of attack, with bugs in similar categories having similar vulnerabilities as well as coverage.

Another key takeaway was the implementation of a benchmark criteria for Smart Contract dataset collation and evaluation. Having a set criterion allowed for consistency and a scientific control measure to be implemented to minimise the possibility of outliers when running the existing Static Analysis tools against each other.

The final significance with immense importance towards the direction and motivations of this thesis project was that of the false-negative findings from the CAPEC results. With medium to high vulnerabilities in Smart Contracts having such poor coverage in existing tools, this would be an interesting point to follow up on, by implementing in the projects tool coverage of bugs which vary from low to high vulnerability, with an increased focus on low to medium impact bugs. Given their existence, such as a gap with a lack of existing tools able to provide coverage for bugs, it would be interesting to explore what these bugs are as well as offering coverage for them.

3.0 Proposed Approach

3.1 Methodology

3.1.1 Methodology Control Flow Diagrams

For the goals and aims of this thesis project to be satisfied, the methodology below was developed. The intention of the methodology was to initially coincide with the existing Static Analysis tools approach of determining which bugs and vulnerabilities to be covered and violation conditions. With the divergent innovation of this project altering the detection approach, by offering increased Bug Attack Theme (BAT) coverage. Building this project using the following modular approach will allow for comparisons and contrasts to academic papers and existing Static Analysis tools in furthering the knowledge and findings in this field of program Analysis. The following methodology was proposed prior to coding of the proposed tool

- Decide on bugs and vulnerabilities which will be detected based on academic papers, existing tools and Solidity documentation [13, 14, 15, 17, 18]
- Categorise bugs and vulnerabilities into Bug Attack Theme (BAT) of Overflow/Underflow, Syntax and DAO
- Evaluate collection of bugs impact and solution towards mitigating or minimising risk of bug on Smart Contract
- Construct Control Flow Diagrams (CFD) for bug collection
- Develop new CFD logic based on variable instances of bugs which do not have existing logic from AST or XPath parse trees from existing tools or academic papers [13, 14]

This methodology was successfully achieved with a collection of bugs and vulnerabilities selected. Their solution and risk Analysis are based on academic papers and existing tools. As well as CFD logic created and derived from either existing AST/XPath logic or newly developed based on multiple variations and examples from previous attacks and code segments.

3.1.2 Methodology Software Tool

The methodology with regards to the proposed Static Analysis tool 'PySolSweep' focuses on the software methodology, since a software solution was aimed to be built. This methodology requires the initial methodology of the Bug Attack Theme (BAT) and Control Flow Diagram (CFD) logic to be integrated into the software package coinciding with the following methodology.

- Develop the tool to allow selection of general Overflow/Underflow, Syntax and DAO as well as DAO withdraw function Static Analysis
- Construct code segments of multiple variations that follow the CFD logic for bugs or violation of precautionary measures
- Construct functions for individual bugs that detect all instances of bugs in code segments test files
- Determine accuracy (Confidence) of each function based on success of detecting test code segment files using Risk Assessment Scale of Low, Medium or High
- Develop Smart Contract security safety rating matrix score, based on bugs risk, confidence and volume to be presented at the end of Static Analysis result
- Develop tool to print results of bug name, Bug Attack Theme (BAT), Line in code, solution, impact and confidence to both command terminal as well as a log (.txt) file
- Construct UI/UX GUI interaction, allowing for visual window tool to select Smart Contract for Static Analysis review
- Develop benchmark criteria based on academic paper [22] existing benchmark criteria, with suitability to the aims and goals of this project
- Collect 100 General and 50 DAO withdraw Smart Contracts from Etherscan coinciding with benchmark criteria
- Conduct testing using Contract collection dataset in a comparative experiment between the proposed tool PySolSweep and existing Static Analysis tools of Remix, sFuzz, SmartCheck, Solidity Scan and Solint.
- Evaluate results from experiment of accuracy and coverage metrics performance Analysis

- Draw insights, progressions and limitations from proposed tool and existing tools experiments results and findings

The project will be written in Python since this fills the limitations of existing tools having too many dependencies as per background research with Linux Operating System restriction and multiple 3rd party libraries. Developing in Python overcomes gaps as derived from the goal of the project. It can be noted that C++ and program Analysis infrastructure of tree building, parser, tree analysers and flow analysers were not in the realm of previous knowledge in undertaking this project. Hence to overcome this limitation, Python was the primary programming language was used in this project. This limitation also opened the door for discussion on future work with using the C++ and parser infrastructure as will be discussed later. The Python libraries required to build this software Static Analysis tool are as follows:

- NumPy: Used to define complex arrays storing scanned results of data structures, types, variables and mapping information used for bug detection verification comparison to the CFD logic.
- Tkinter: The GUI toolkit enabling user interactions and user experience to select and view the result of Static Analysis on their Solidity Smart Contract

3.2 Attacks

The attacks focused on in this project include the Bug Attack Theme (BAT) of Overflow/Underflow, Syntax and DAO. Each BAT contained bugs and vulnerabilities which could be classified within its sphere. To be able to construct CFD's as a pipeline for the software Static Analysis tool, each bug was evaluated by firstly using academic papers reviewed and Solidity documentation to generate an overview of each of the three BAT's which included the bug, solution and risk. This construction would be important in understanding the bug's behaviour to help construct CFDs, as well as later in the project's software tool in what output is written to the terminal/log file with regards to each bug's characteristics.

3.2.1 Overflow/Underflow Attack

Overflow/Underflow bugs detected included those covered in a mixture of existing tools and academic papers [13, 14, 15, 17, 18]. However, some bugs which were not covered in existing tools but were found from Solidity documentation included division before multiplying, where using division could cause loss of numerical precision in Solidity. Unary operations of unintended increments and Type Inference of explicitly declaring numerical values were also new bug coverage which did not exist in previous tools Static Analysis.

An overview of the Overflow/Underflow bugs covered in the project is summarised in **Table 2**.

Table 2: Overflow/Underflow Coverage Table

Bug	Detail	Solution	Impact
Safe Math	This check catches if a Smart Contract is defined using the Safe Math Library	Safe Math library will catch errors and throw exceptions	Medium
Integer Operations	This check catches if a Smart Contract defined the uint data type without the Safe Math library function	Safe Math library will catch errors and throw exceptions if an Overflow or Underflow condition exists in uint data types	High
Loop Condition	This check catches if a Smart Contract uses arithmetic operations [+, -, *, /, %]	Safe Math library has functions of add, sub, mul, div and mod to minimise case of Overflow or Underflow exploit without loss of precision	Medium
Division Before Multiplication	This check catches if a Smart Contract has arithmetic operations of division before multiplication	Use multiplication first, as division first causes loss of precision in operations	Medium
Unary	This check catches if a Smart Contract contains =+, =- or =* which could be intended as =+, =- or *=	To minimise misconception use +=, -= or *= operations	Low
Type Inference	This check catches if a Smart Contract is defined using var instead of numerical data type uint	Should explicitly declare uint data type to avoid unexpected behaviours	Medium

3.2.2 Syntax Attack

The Syntax bugs added to the coverage Bug Attack Theme (BAT) focus more on unintended or exploiting the existing confines and rules within the Solidity programming language for Smart Contracts. The Syntax BAT class coverage includes 24 bugs and vulnerabilities. However only 7 are covered in existing Static Analysis tools. These 7 include compiler, transfer, Tx origin, loop function, block timestamp, delegate call and function visibility. The proposed tool extends on the existing loop gas vulnerability, since loop gas usage is dependent on the condition, the proposed tool additionally checks that loop conditions are Static and not dynamic to ensure gas usage does not exceed the intended amount for any transaction.

The other 17 Syntax bugs were again derived from Solidity documentation as well as academic papers [13, 14, 15, 17, 18] as bugs which coincide with a Syntax attack on Smart Contracts. These 17 bugs that were not covered with existing tools vary from low to high impact and include Boolean constant, array length, array zero, map struct delete, initial storage variable, assembly shift, self-destruct ,bytes balance equality, block variable/number/gas calls, owner power, constructor, local and state variable shadowing and fallback.

The innovation in this project not only adds 17 new bugs to be increase coverage in a syntax attack, but also draws from different existing tools and academic papers [13, 14, 15, 17, 18] to increase the coverage within bugs itself and the increased variation that the initial 7 previously covered have changed and other different ways they could be exploited as demonstrated in **Table 3.**

Table 3: Syntax Existing Coverage Table

Bug	**Detail**	**Solution**	**Impact**
Compiler	This check catches if a Smart Contract is defined using the ^ operator for compiler version.	Best practice to use Static rather than dynamic compiler version as future versions could have unintended effects	Medium
Transfer	This check catches if a Smart Contract contains call.value or send value methods	Use transfer function instead of send/call operation as they do not capture transaction fails to minimise vulnerability	Low
Tx Origin	This check catches if a Smart Contract contains TX origin function	Use msg.sender instead of TX. Origin to minimise vulnerability. tx.origin is vulnerable for authentication as it can be manipulated to be equal to an owner address hence pass the require tests	High
Function Visibility	This check catches if a Smart Contract contains functions with unknown visibility	Use at least minimum public/private specifier when defining function to minimise vulnerability	High
Block Timestamp	This check catches if a Smart Contract contains block.timestamp for randomness	Avoid block.randomness for randomness to minimise DoS vulnerability	Low
Delegate Call	This check catches if a delegate call is made, potential for parity sig wallet attack	Avoid Delegate Call this can lead to unexpected code execution vulnerability	Medium
Loop Function	This check catches if a function call is made within a for or while loop	Avoid Function Call in For/While Loop possible DoS vulnerability	Medium

The Syntax bugs 17 new bugs to increase coverage in a syntax attack were constructed in the following **Table 4**.

Table 4: Syntax Additional Coverage Table

Bug	Detail	Solution	Impact
Boolean Constant	This check catches if a Smart Contract incorporates Boolean Constance or Tautology conditions	Should verify that Tautology is not intended as well as Constance is not intended	Low
Array Length	This check catches if a Smart Contract defined an array with a Static length	Should increase array length as array grows and storage needed	Medium
Address Zero	This check catches if a Smart Contract function does not check that the address is zero using address(0), 0x0 or address(0x0)	Check address is not zero using require and address variable reduce likelihood of interaction with a null address	Medium
Map Struct Delete	This check catches if a Smart Contract either defines a map struct as a different data type or uses delete keyword for mapping delete which does not not delete the entire mapping only deletes entry	Should Use the same data type key as defined in struct for mapping. Use lock technique mechanism to disable mapping structure if needed to remove	Medium
Initial Storage Variable	This check catches if a Smart Contract includes struct variables which are not set when using struct	Should Immediately initialise storage variables could be overridden	High
Assemble Shift	This check catches if a Smart Contract includes an assembly shift that has parameters mismatched in their order	When using shr assembly shift if first position is a variable and second is constant, this bit shift is usually unintended	Medium

Self-Destruct	This check catches if a Smart Contract contains a self-destruct with address or a function that is public and uses self-destruct	When an address in self-destruct address is not used, it could send ether to an attacker Contract. If using self-destruct restrict access to function as not public	High
Bytes	This check catches if a Smart Contract contains bytes array instead of using bytes	Use bytes instead of byte array as this could grow and access unintended storage	High
Balance Equality	This check catches if a Smart Contract double equals for evaluation of a balance variable	Use comparative statements instead of double equals to minimise vulnerability	Medium
Block Variable	This check catches if a Smart Contract contains block.timestamp/gas limit or difficulty	Potential leaky PRNGS rely heavily on past block hashes future vulnerability	Low
Block Number	This check catches if a Smart Contract contains block.number	Check function when getting current block number could be invoked by an attacker for malicious intent	Low
Block Gas	This check catches if a Contract contains for/while loops which condition uses the length of an array or object to iterate over	Avoid loop of unknown size that could grow and cause DoS vulnerability	Medium
Loop Function	This check catches if a function call is made within a for or while loop	Avoid Function Call in For/While Loop possible DoS vulnerability	Medium
Owner Power	This check catches if a Contract basis functions control and execution on the owner. Or a modifier function is used to define an owner.	Owner private key at risk of being compromised do not base function control on owner or use an owner modifier function	High
Constructor	This check catches if a Contract defines multiple constructors either through constructor or a function constructor. Checks whether the same variables are defined over multiple constructors could be overwritten.	Use a single constructor to initialise a Contract; the second constructor will be ignored. Use single constructor and initialise variables once in constructor	Low

Local Variable Shadowing	This check catches if a Contract contains the phenomenon of local variable shadowing. This includes the local variable shadows, an instance variable in the outer scope based on the modifier, struct, function, constructor and mapping.	Consider renaming local function variables to mitigate unintended local variable shadowing or consider not redefining Contract local variables unless intended to.	Low
State Variable Shadowing	This check catches if a Contract contains the phenomenon of state variable shadowing. This includes using and redefine inherited variables from the parent Contract in the child Contract	Solutions include assigning a Parent Contract prior to the child Contract. Define inherited parent Contract variable in Constructor. Same variable name from parent redefined use different variable name. Parent Contract variable never assigned, assign in parent Contract to prevent	High
Fallback	This check catches if a Contract contains an external Fallback function for transfer of ether. Without being marked as payable Contract could through error and be inactive without this component	Mark Fallback function with payable otherwise Contract cannot receive ether	Medium

3.2.3 DAO Attack

The DAO bugs were added to the coverage Bug Attack Theme (BAT) class, with 5 bugs covered. Compared to the existing tools which only detected one variance of DAO Reentrancy which focused on a withdraw function not checking if the withdraw amount exceeded the balance amount. From papers reviewed it was evident that other DAO bugs could be exploited, hence 4 additional bugs and vulnerabilities were added including Contract lock, balance state variable, external call and the precaution measure check-effect-interact pattern. The check-effect-interact pattern in particular was a defence mechanism for DAO as described in academic papers [15], hence its inclusion in the constructed DAO BAT summary **Table 5**.

Table 5: DAO Coverage Table

Bug	Detail	Solution	Impact
Contract Lock	A Contract does not contain a lock modifier for Reentracy attack. As well as whether conditions of require, true condition guard for Reentracy conditions by checking external calls that are unprotected.	Use a blockreentrancy Contract lock mechanism so only a single Contract function is executed	Medium
Reentracy Require	Contract with the withdraw function conducts a require verification of amount and balance state variable to ensure funds are not in correctly extracted by an attacker	Condition needs this to check require balance and amount first before any operations in withdraw function	Medium
Balance State Update	Contract with the withdraw function conducts an update to the Balance state variable prior to any operations such as call, send or transfer.	Condition to Update state variable before call to prevent Reentracy multiple calls from attacker	High
External Call	Contract that calls an external function from another Contract is marked as either trusted or untrusted. If untrusted this could be vulnerable to an attack invoked by the adversary.	Be aware that subsequent calls also inherit untrust state. Unknown trust, label function either trusted/untrusted	High
Check Effect Interact Pattern	Contract with the withdraw function conducts the Checks-effects-interactions pattern when withdrawing funds from the balance. This pattern can ensure that all prerequisites are met before executing the entire withdrawal. This pattern will prevent reclusive calls by managing the Reentracy state.	Incorporate the Check-Effect-Interacts pattern, ensure that order is correct. Including all three components will function as a Reentracy guard. However, if out of order, the Contract withdraw function could still be vulnerable to DAO Reentracy attack.	High

3.3 Control Flow Diagrams

The Control Flow Diagram (CFD) logic illustrates an equivalence to Abstract Syntax Trees (AST) used in existing tools architecture for Static Analysis. Having the details of Bugs and their respective Bug Attack Themes (BAT) enabled for development of the proposed software tool, as well as achieving the limitation of 23 bugs and vulnerabilities which were not detected by existing tools' parse trees could have logic implemented for detection for both this project and future work.

3.3.1 Existing Bugs & Vulnerability Detection Logic

For bugs and vulnerabilities 11 existed across academic papers reviewed [13, 14, 15, 17, 18] as well as current Static Analysis tools. The 11 constructed Control Flow Diagrams derived from logic of violations or precautions not taken. This existing logic includes for example as illustrated in **Figure 6** for the simple function visibility Syntax BAT bug, forks and modules derived from Solidity Documentation as well as existing tools' parse tree logic for paths in violation

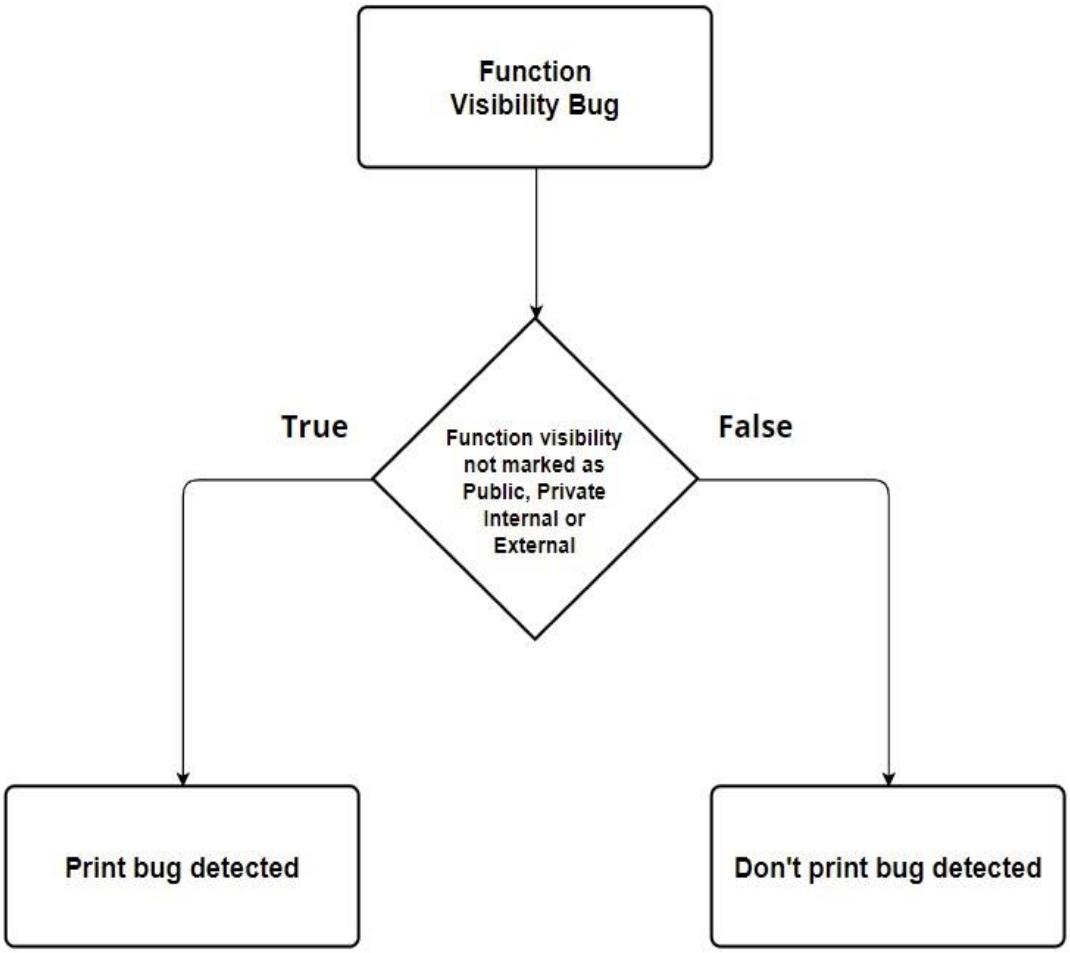


Figure 6: Existing Bug CFD Violation Paths Example

Whilst for existing bugs which were covered, but from academic papers reviewed [13, 14, 15, 17, 18] could have additional coverage in terms of more variations. Additional forks and modules to the CFD were integrated. This includes for example the for/while loop gas usage bug. In which only strict comparative operators '>=' or '<=' were bugs covered in existing tools. Academic papers [14] reviewed also found the 'uint' iterator or '.length' as a loop condition could adversely increase gas usage in transactions, hence a vulnerability. This addition to existing logic is evident in **Figure 7**.

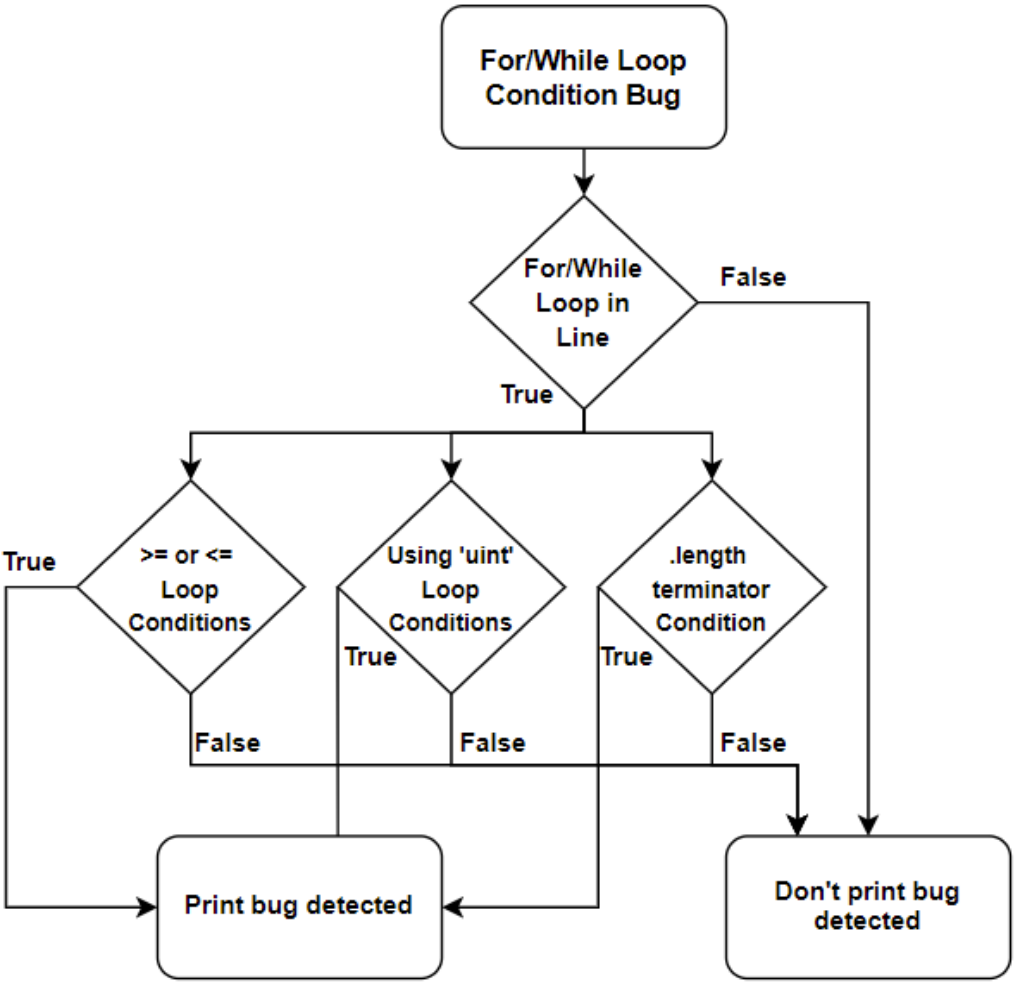


Figure 7: Existing Bug Extended CFD Violation Paths Example

3.3.2 New Bugs & Vulnerability Detection Logic

For the limitation of 23 bugs and vulnerabilities which had excerpts of examples and behaviour from academic reviewed papers [13, 14, 15, 17, 18] as well as Solidity documentation, Control Flow Diagrams were constructed by approach of detecting those conditions, violations and precautionary measures listed in reviewed material and Solidity documentation. This construction of logic addressed the gap of a lack of detection logic for Bug Attack Theme (BAT) bugs which were not covered by existing Static Analysis tools. An example from the 23 new CFD's below includes the Check-Effect-Interact search, which without this pattern in a Smart Contract withdraw

function can cause the withdraw function to be liable from a DAO Reentracy attack, which is maliciously recursively transferring funds is illustrated in **Figure 8**.

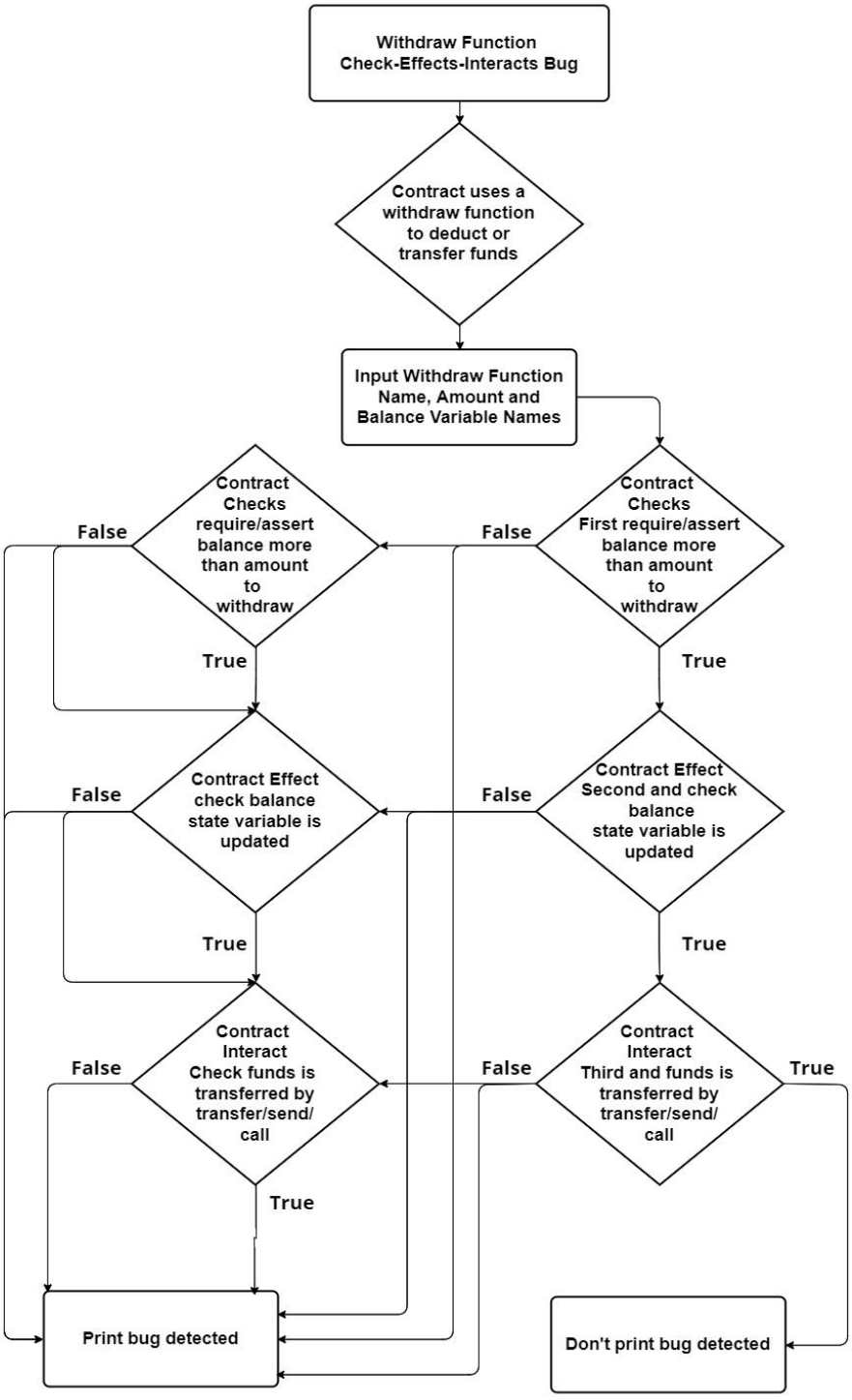


Figure 8: New Bug CFD Violation Paths Example

3.3.4 Control Flow Diagram Implementation

The implementation of the 35 Control Flow Diagrams (CFD) paved the way for the logic architecture to be adapted by each Python function in the code. Each fork and module in the control flow diagram would represent the search method, violation conditions and steps taken in each bug detection that occurred in my project's Python based Static Analysis tool. This implementation can be summarised below **Figure 9**.

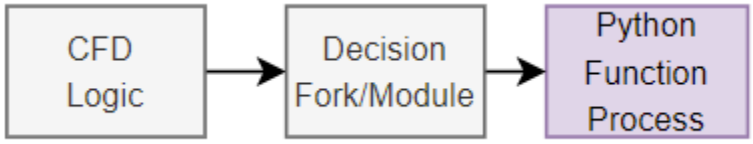


Figure 9: Proposed CFD Implementation Approach

3.3.5 Control Flow Diagram Logic Limitations

Reflecting on the execution of the methodology thus far, it is evident of one of the projects negative points of not using a parsing software infrastructure to translate this produced CFD logic into an XML Abstract Syntax Tree (AST) for Static Analysis using and XML analyser due to be out scope of this project. However as future work this could be conducted, in line with industry Static Analysis tools using XML Analysis programs. Another limitation includes that of verification of logic, the 11 of the CFD's are based on existing tools and academic papers reviews [13, 14, 15, 17, 18]. Whilst the 23 newly proposed CFD logic, will need to be more thoroughly evaluated with volume and robust testing. By more than just one singular experiment that was later conducted in this project for verification, the CFD logic indeed holds true. Hence, future work of multiple large-scale experiments implementing the CFD logic on peer verified instances of those bugs could be done, to increase the validity and accuracy of the constructed Control Flow Diagrams. These limitations, however, do not detract from the goal of this project of producing new CFD logic for bugs within the three Bug Attack Themes (BAT), in offering increased attack coverage.

4.0 Implementation

The software product as proposed by the project namely '*PySolSweep'* could be developed and implemented using Control Flow Diagram (CFD) extracted forks and modules when coding the software tool. With the goal to overcome the existing solution of allowing cross-platform usage being Python based this will be achieved. Overcoming the limitation of a lack of Bug Attack Theme (BAT) coverage by increasing and focusing on bugs which instigate three types of BAT's of Overflow/Underflow, Syntax and DAO. With the final limitation aimed to be satisfied that of a poor UI/UX with little interface, no log to view Static Analysis results and a lack of a suggested solution. These listed limitations were overcome in developing and constructing the Python native Solidity Static Analysis software tool.

4.1 Bugs Code Segment Generation

To overcome one of the limitations identified (*3.3.5)* of adding credibility to the proposed CFD's as well as cross-check that each bug detection function achieves its aim of detecting its desired bug or vulnerability, bug code segments were generated. This process involved for each of the 33 types of bugs and vulnerabilities, a test Smart Contract was generated containing multiple variations of that bug, based on the violation path in the CFD. Furthermore to ensure that non-violation paths were not detected as a bug, these non-violating CFD paths were also added to each test Contract to ensure false positive conditions did not occur. The intention of this was to address the aim of the project's goal in matching accuracy (non-false-positive rates) of existing Static Analysis tools. A total of 53 test Smart Contract code segments were constructed for each of the 33 bugs, with multiple variations as well as non CFD violation paths followed. This implementation is portrayed in an example excerpt of one of the test Smart Contracts below, which focused on the Syntax BAT of local variable shadowing, following multiple bug and non-bugs CFD paths for evaluating detection accuracy of the project's tool in **Figure 10**.

```
function alternate_sensitive_function() public {
    address owner = msg.sender;
    require(owner == msg.sender); //BAD
}
function alternate_sensitive_function() public {
    uint owner = msg.sender;
    require(owner == msg.sender); //BAD
}
function alternate_sensitive_function() public {
    address own = msg.sender;
    require(own == msg.sender); // GOOD
}
function sensitive_function(address owner) public
    onlyOwner{
    require(owner == msg.sender); //BAD
}
function sensitive_function(address send) public {
    require(send == msg.sender); //BAD
}
function sensitive_function(uint own) public {
    require(own == msg.sender); // GOOD
}
```

Figure 10: Example Code Segment Generated for Local Variable Shadowing Bug

4.2 Code Structure

4.2.1 Graphical User Interface

The proposed tool offered Static Analysis of the general Bug Attack Themes (BAT's) of Overflow/Underflow, Syntax and DAO as well as DAO withdraw function. The GUI was split into two modes of general and complex withdraw Static Analysis. Allowing for two selective options for Smart Contracts to be scanned in accordance with the CFD logic. This GUI interface was created using the Tkinter Python library, with the form allowing for text fields to be filled. With general Smart Contracts requiring the file location within a 'Verify' sub-folder. With the DAO withdraw function Analysis selection also requires file location as well as withdraw function name, amount and balance variable name to be manually entered. Reflecting on the second detection mode of the DAO withdraw function, in future work implementing the same CFD logic and approach of BAT coverage through a Symbolic Execution infrastructure for Static Analysis would further minimise the need for user input of variable names involved with the DAO withdraw function Static Analysis.

4.2.2 Bug Detection Functions

To detect bugs and vulnerabilities within the initial code segment test files, 33 functions were created to detect all bugs as covered in this project. Each function followed a systematic approach when amalgamating logic of forks and modules from the Control Flow Diagrams (CFD's) produced of violation paths. The steps taken by each function to achieve this CFD logic implementation is summarised as follows:

- Passed opened log file parameter to be write results to as well as Smart Contract file in text format to be analysed
- Conducted first scan line by line of Smart Contract text file using a for loop, storing global variable names, structs, mappings, data types and data structures in a NumPy array
    - If type and variable name both relevant to a bug or vulnerability, the pair was stored in a key-value dictionary data structure to use as a link if needed for checks of CFD violation

- Constant violation conditions or statements that were Static were assigned to variables prior to second scan for comparative checks
- Conducted second scan of Smart Contract line by line, comparing it against CFD Static violation conditions through if or switch statements inside this for loop scan
    - If array or dictionary of global variables/mapping/struct were relevant, this was iterated over in second scan against CFD violation conditions
- Incremented number count for each BAT if bug detected was within the realm of that BAT, to be presented in result summary
- If bug existed global score variable was increased as per score logic (*4.2.3*)
- After each function completed its double scan Static Analysis, results were written to a log file as well as output to users terminal for viewing.

The complexity of functions which each conducted a search for each CFD violation path varied for example from simple Overflow/Underflow BAT of using '+' instead of the error catching SafeMath library '.add'. To more complex violation paths such as for example the Syntax BAT of local variable shadowing which required referencing stored first scan dictionaries of substring operations to attain data types and global variable names against local function to check if the inner scope shadows the instance variable in an outer scope. This approach to detect bugs and vulnerabilities following CFD violation paths is illuminated in **Figure 11**.

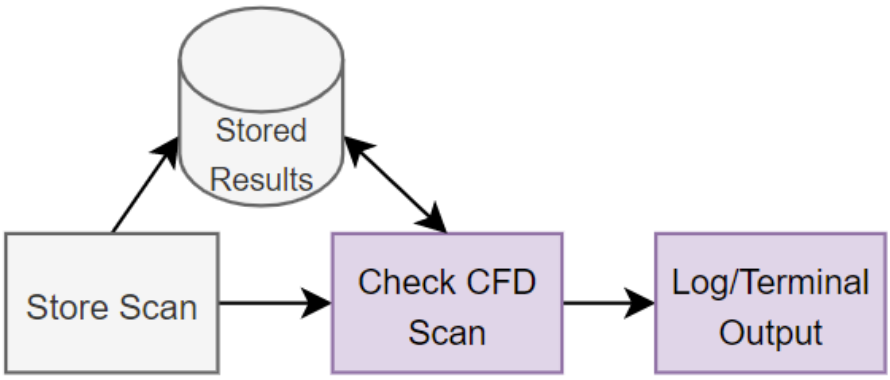


Figure 11: Function Bug/Vulnerability Detection Process

Overall, each of the 33 bug and vulnerability scanning functions achieved the project's aim of implementing the violation paths produced from the Control Flow Diagrams. To offer increased BAT coverage against attacks, relative to existing tools as well as verify using code segment test Contracts that the CFD logic can attain credibility in detecting bugs in a controlled environment.

4.2.3 Contract Safety Rating

The project aimed to produce a Contract safety rating score after a Static Analysis scan is complete, in which the Smart Contract client can decide whether to deploy or address bugs and vulnerabilities found by the scan before deployment on the blockchain. This was achieved by implementing a guideline for calculating score in which a global score variable was passed from function to function. Each time a bug or vulnerability was found, the score was incremented in accordance with the Risk-Confidence matrix. With risk already determined for each BAT bug/vulnerability the confidence was determined by the functions accuracy on the test code segments. Higher accuracy correlated to a high confidence, whilst lower accuracy relative to all detection functions resulted in a low confidence for that bug/vulnerability. Smart Contracts which attained a higher volume of critical/moderate rated bugs or vulnerabilities received a lower Contract safety score (Appendix D). Whilst Smart Contracts with a lower volume made up mostly of minor/moderate bugs or vulnerabilities attained a higher Contract safety score. The constructed Risk-Confidence matrix is illuminated in **Figure 12**.

RISK ↓ / CONFIDENCE →	LOW (1)	MEDIUM (2)	HIGH (3)
LOW (2)	MINOR – 2 –	MINOR – 4 –	MODERATE – 6 –
MEDIUM (4)	MINOR – 4 –	MODERATE – 8 –	CRITICAL – 12 –
HIGH (6)	MODERATE – 6 –	CRITICAL – 12 –	CRITICAL – 18 –

Figure 12: Constructed Bug Classification Risk-Confidence Matrix

4.3 Static Analysis Program

4.3.1 General Scan

The first Static Analysis mode produced was that of a general scan, which covers all three Bug Attack Theme (BAT) classes of Overflow/Underflow, Syntax and DAO. The use-case involves a Smart Contract being placed within the '*Verify*' subfolder. The file location would then be input as 'Verify/filename.txt', with the Solidity code transformed into a Python readable text format. The '*Start Scan Analysis'* UI button would be selected and the previously defined process (*4.2*) of Static Analysis of the Smart Contract would be executed. Once completed, the results were displayed both in the terminal as well as an output text file namely '*bugreport.txt*'. The results contained all requirements as per the methodology including the bug detail, line in code, potential attack, solution which overcomes existing limitation of no bug-corrective advice, risk, confidence, BAT summary and a Contract safety score rating at the end of the bug report. The general Static Analysis scan bridged the gap of a lack of a bug solution as well as provided increased BAT coverage, which were limitations of existing Solidity Static Analysis tools. This general scan process is visually illuminated as evident in **Figure 13**.

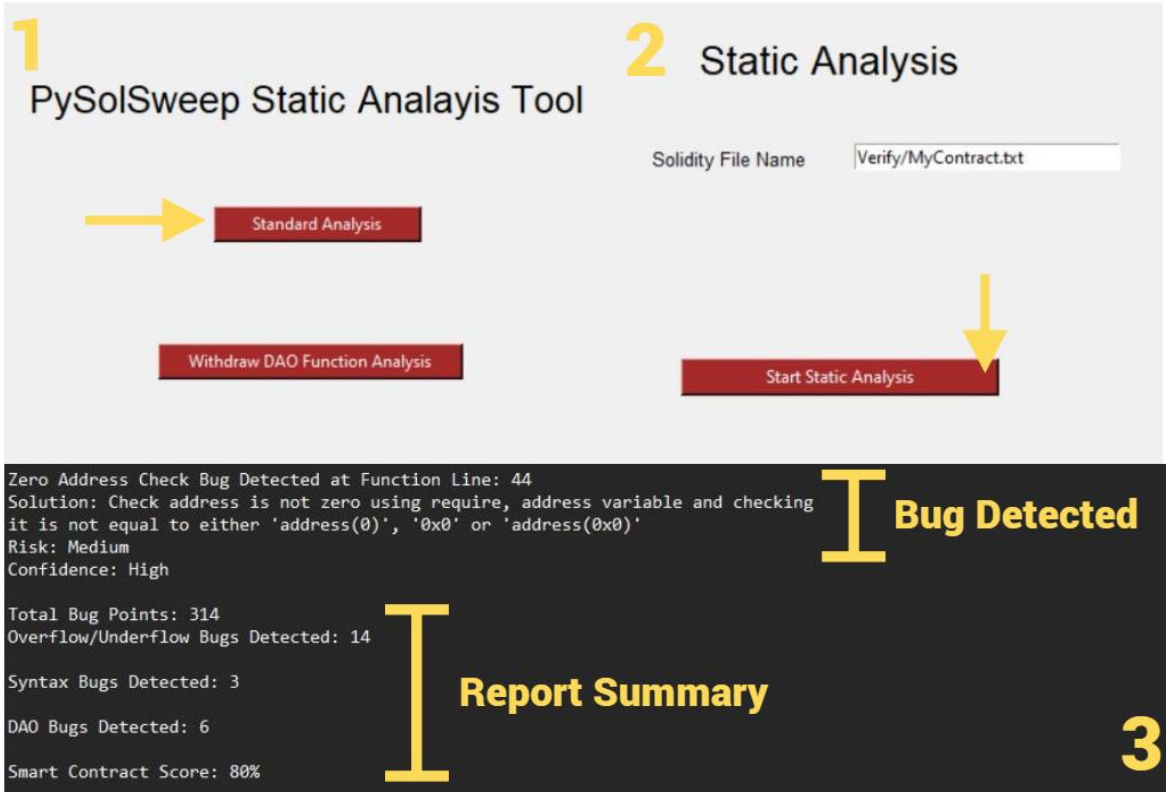


Figure 13: PySolSweep General Static Analysis Scan Demo

4.3.2 DAO Withdraw Scan

The second Static Analysis mode produced was that of a DAO withdraw function scan, which covered DAO bugs and vulnerabilities specific to a Smart Contracts withdraw/transfer function. The use-case follows similar to the general scan with the Smart Contract again placed within the '*Verify*' subfolder. One of the limitations of this project is observed next, as the client is required to fill the Static Analysis form with the withdraw/transfer function name, balance state variable and withdraw/transfer amount variable names. Again, as previously stated (*4.2.1*), future work implementing this project's approach in a symbolic execution infrastructure would mitigate the need to manually enter this metadata. Moreover, the benefits of doing so yield a 400% increase in DAO BAT coverage than existing Solidity Static Analysis tools from 1 to in the projects tool 5 bugs, vulnerabilities and precautions detected in this scan mode. The output to the terminal and log file mimics that of the general scan (*4.3.1*) results. The DAO withdraw Static Analysis scan mode process can be observed in **Figure 14**.

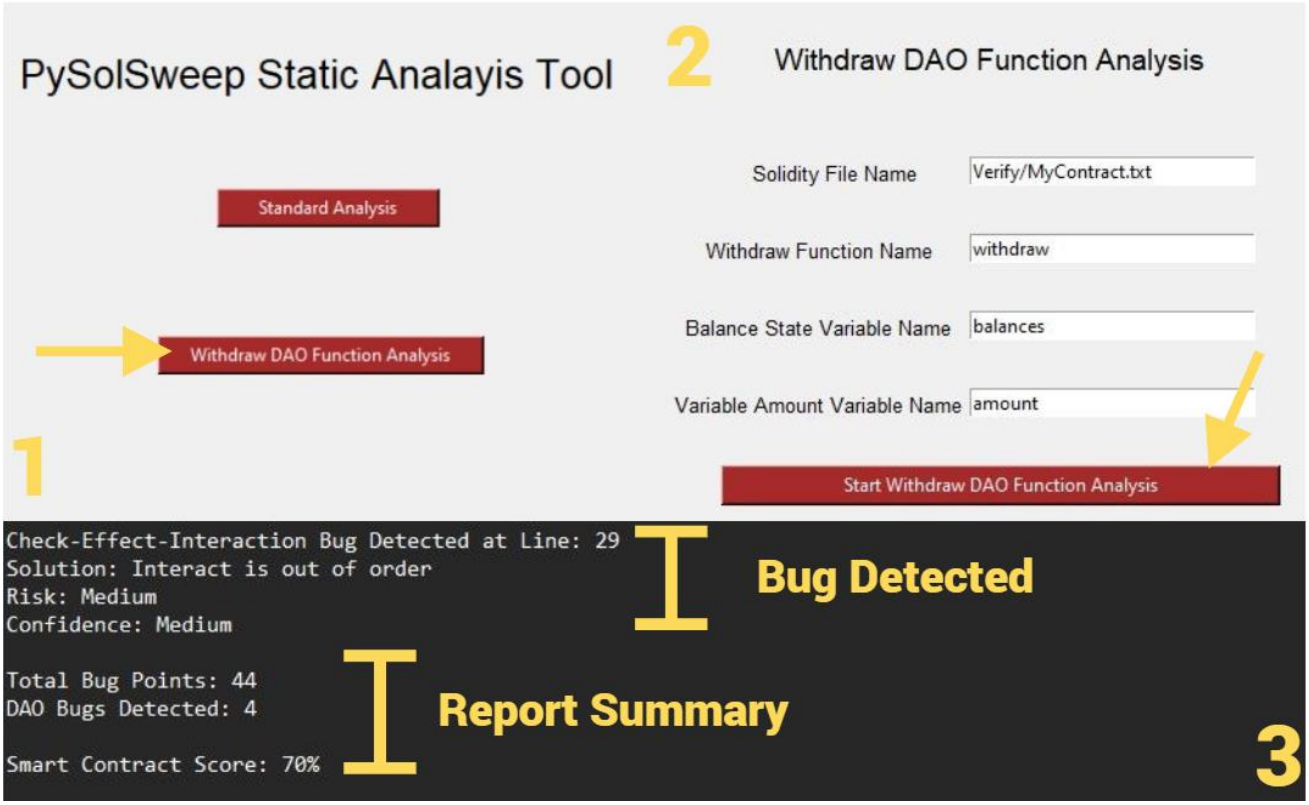


Figure 14: PySolSweep DAO Withdraw Function Static Analysis Scan Demo

5.0 Evaluation

The process of evaluation was conducted to both interpret credibility of the Control Flow Diagram (CFD) logic integrated into the BAT (Bug Attack Theme) functions in the project's proposed tool '*PySolSweep'*. But, as well as observe whether the limitations of existing tools having poor BAT could be overcome. These goals were addressed through an experiment which evaluated '*PySolSweep'* against current Solidity Static Analysis tools performance on deployed Smart Contracts.

5.1 Testing Benchmark Criteria

The evaluation of Static Analysis tools required currently deployed Smart Contract code to be Statically analysed. In order for this to be achieved, a Smart Contract dataset collection would be required. Coinciding with academic papers steps for collection, all Solidity Smart Contract code was sourced from Etherscan. Etherscan, which previous academic papers [13, 22] used, is a blockchain explorer for the Ethereum network, enabling the Smart Contracts Solidity source code for transactions, libraries or services to be publicly viewed. With millions of Smart Contracts passing through Etherscan, all diverse in nature and behaviour. It was important for quality results of the aim of the project to be evaluated whether PySolSweep's approach of increased BAT detected more bugs whilst maintaining accuracy than existing Static Analysis tools. This aim was achieved by firstly generating a Benchmark criterion. The justification of this criteria derived both from previous academic papers [13, 14, 22], as well as the intention to protect Smart Contracts from BAT's of Overflow/Underflow, Syntax and DAO. Which could propagate within a diverse Smart Contract. The following benchmark criteria was used to produce the Solidity Smart Contract dataset for testing **Table 6 & 7**.

Table 6: General Contract Benchmark Criteria

Criteria	Justification
Minimum 200 Lines of Code	Allow for sufficient complexity to occur
MIT licence SPDX identifier	Permission for usage in testing
Numerical Operations	Bugs related to Overflow/Underflow
For/While Loop Operations	Multiple BAT coverage
Gas Usage	Multiple BAT coverage
Global Variables	Multiple BAT coverage
Constructor Defined	Multiple BAT coverage
Structs Data Structure	Multiple BAT coverage
Mapping Data Structure	Multiple BAT coverage
Functions Public/Private/External/Internal	Multiple BAT coverage
Function Complexity Minimum 5 Lines of Code	Simple get and set functions do not contain significant vulnerabilities

Table 7: DAO Withdraw Function Contract Benchmark Criteria

Criteria	Justification
Withdraw/Transfer Function	Allow for Analysis for DAO BAT this type of function is most vulnerable to exploit
Remove non relevant Line of Code	Reduce testing time for other functions which do not interact with withdraw/transfer function

5.2 Smart Contract Dataset

By applying the benchmark criteria on Etherscan's Ethereum blockchain explorer, a total of 150 Solidity Smart Contract's code were acquired. A total of 100 coincided with the general benchmark criteria to investigate general BAT coverage. Whilst 50 Smart Contract codes were allotted to the DAO withdraw function BAT coverage. Reflecting on this process of collecting a Smart Contract code dataset. Multiple Contracts did incorporate external libraries for usage and interactions. To ensure consistency and focus on the Smart Contract itself rather than its libraries used, the library code was omitted and only its usage in the Smart Contract was retained. The intention for this was the repetition of common interfaces used in constructing a Smart Contract such as ERC, SafeMath, IERC, Ownable, UniSwap, Strings and Context. These were visible in the majority of deployed Smart Contracts. Since their implementation was not in the scope as it would increase testing time this code was omitted. Whilst their functions, libraries usage and integration within the main Smart Contract was retained to be Statically analysed.

5.3 Comparative Tools

5.3.1 Solidity Scan

The Static Analysis tool Solidity Scan performs program Static Analysis on Solidity Smart Contracts. The requirements to run this tool include Truffle and Hardhat which both are pipelines for the EVM (Ethereum Virtual Machine) to run [23]. These multiple dependencies were identified as a limitation and hence a gap to overcome in this project.

5.3.2 SmartCheck

SmartCheck being another Solidity Static Analysis tool was released in May 2018. This Static Analysis tool required the node manager package as well as a Linux Operating System environment to run. For this experiment a Linux virtual machine was set up using the VirtualBox hosting software tool, further elaborating the lack of cross-operating compatibility. Furthermore, being released in 2018 and last updated in 2019, this as per background research (*2.0*) has seen new bugs, vulnerabilities and attack surface [24].

5.3.3 sFuzz

The tool sFuzz (Contract Guard) was a web browser based Static Analysis tool, which accepted correctly syntactically formatted Solidity code to be analysed. This was the most usable tool; however the caveat was that the tool could only conduct Analysis with Contracts less than 250 lines of code. Given the benchmark criteria this was sufficient. However, it lacked scalability to longer and more complex Smart Contracts on Etherscan. Furthermore, similar to SmartCheck the tool was created in 2019, leaving it behind in new bugs and vulnerabilities subject to Smart Contracts[25].

5.3.4 Solint

Solint was also a Solidity Static Analysis software program, which was released in 2019 also faced the same problems of lacking bug coverage. Such as the DAO check-effect-interact pattern which was a safety precaution introduced by Solidity documents in 2020 [26]. Similar to Solidity scan, Solint was dependent on Truffle as well as the Remote Procedure Call (RPC) client Ganache to read from a local Solidity file [27].

5.3.4 Remix

The final tool Remix is a node package manager Static Analysis tool for Solidity Smart Contracts. This tool was released in 2020, being the most up to date tools evaluated in the experiment. This tool required a Linux or Macintosh environment to analyse Smart Contracts. Hence reinforcing the dependency limitation outlined in this project's aims to overcome [28].

5.3.5 Bug Attack Theme Proposed Coverage

Since the experiment focuses on the coverage of three major Bug Attack Themes (BAT) of Overflow/Underflow, Syntax and DAO. The total number of bugs and vulnerabilities that each of the tools prior to conducting the experiment should be able to detect is outlined in **Table 8**.

Table 8: Static Analysis Tools BAT Coverage

Tool	BAT-1	BAT-2	BAT-3
Solidity Scan	*2*	*8*	*1*
SmartCheck	*3*	*10*	*1*
sFuzz	*2*	*14*	*1*
Solint	*2*	*6*	*1*
Remix	*2*	*8*	*1*
*PySolSweep**	*6*	*24*	*5*

**BAT-1: Overflow/Underflow, BAT-2: Syntax, BAT-3: DAO*

5.4 Test Environment

5.4.1 Experiment Independent and Dependant Variables

The test environment incorporated setting up the 5 existing Static Analysis tools (*5.3.5*) alongside the project's proposed tool '*PySolSweep'*. The required dependencies of Truffle, Ganache and Hardhat as well as multiple VirtualBox virtual machines running Linux OS were set up. The methodology for the experiment included the independent variable being the 100 general Smart Contracts and the 50 DAO withdraw/transfer function Smart Contracts Solidity code. Whilst the dependent variable was the bugs detected and false positive rate of each detection being classified into each of the BAT of Overflow/Underflow, Syntax and DAO. The verification for a false positive detection, a cross-check was conducted against the logic of the adjacent Control Flow Diagram (CFD) logic. This inferred to each bug or vulnerability detected, if it failed to follow a violation path in the CFD logic, then this would be counted as a false positive BAT detection. These were manually checked by following the violation paths of the projects proposed CFD's against the bug if it held true.

5.4.2 Experiment Limitations Discussion

A limitation of annually verifying the bug or vulnerability against the proposed CFD's logic, placed huge assumptions that the CFD logic was universally correct. The counter argument to this limitation was that by running multiple Static Analysis tools, if a general consensus of a bug or vulnerability matched the CFD logic across multiple tools. Then, this general consensus would reinforce that the logic as well as increase credibility of the proposed CFD logic holding true universally, given multiple other Static Analysis tools reached the same conclusion for detecting a bug or vulnerability. This general majority consensus to verify the correctness of the proposed CFD notion is further exemplified in **Figure 15**.

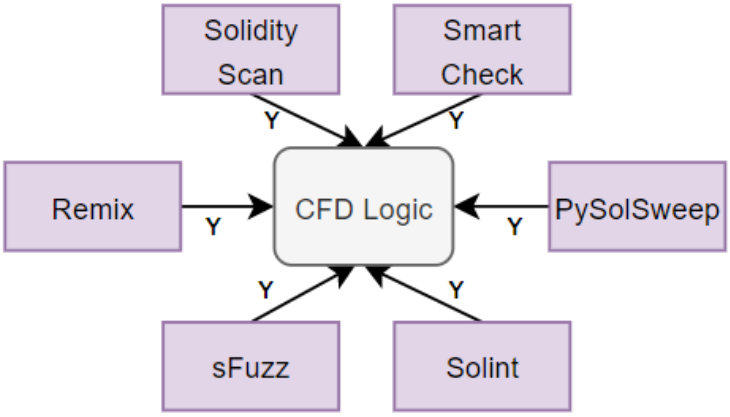


Figure 15: Proposed CFD Bug Logic Achieving Total General Consensus

5.4.3 Experiment Hypothesis

The hypothesis proposed for the experiment incorporated that of the proposed tool PySolSweep, sFuzz and SmartCheck offering higher detection rates. Hence higher Bug Attack Theme (BAT) coverage than the other tools. This notion is reinforced by the proposed BAT bugs and vulnerability coverage identified prior to testing (*5.3.5*). Furthermore it was also hypothesised that PySolSweep would detect more bugs and vulnerabilities than any tools, especially the DAO BAT bugs due to a higher BAT coverage (*5.3.5*). The final hypothesis that was made was that PySolSweep would have slightly degraded accuracy, given the implementation strayed away from the general symbolic execution or ANTLR parsing infrastructure which would have higher accuracy than the double Python scan, store and compare approach taken in this project.

5.5 Results

The experiment of running the 150 Etherscan Solidity Smart Contracts, through each of the 5 existing and project proposed tools was completed over a time period of 6 weeks. Each Smart Contract was run in unison with each of the investigated Static Analysis tools to allow for the CFD correctness consensus to be also measured (*5.4.2*). The following results were produced as of concluding the experiment testing period.

5.1.1 Detection Accuracy

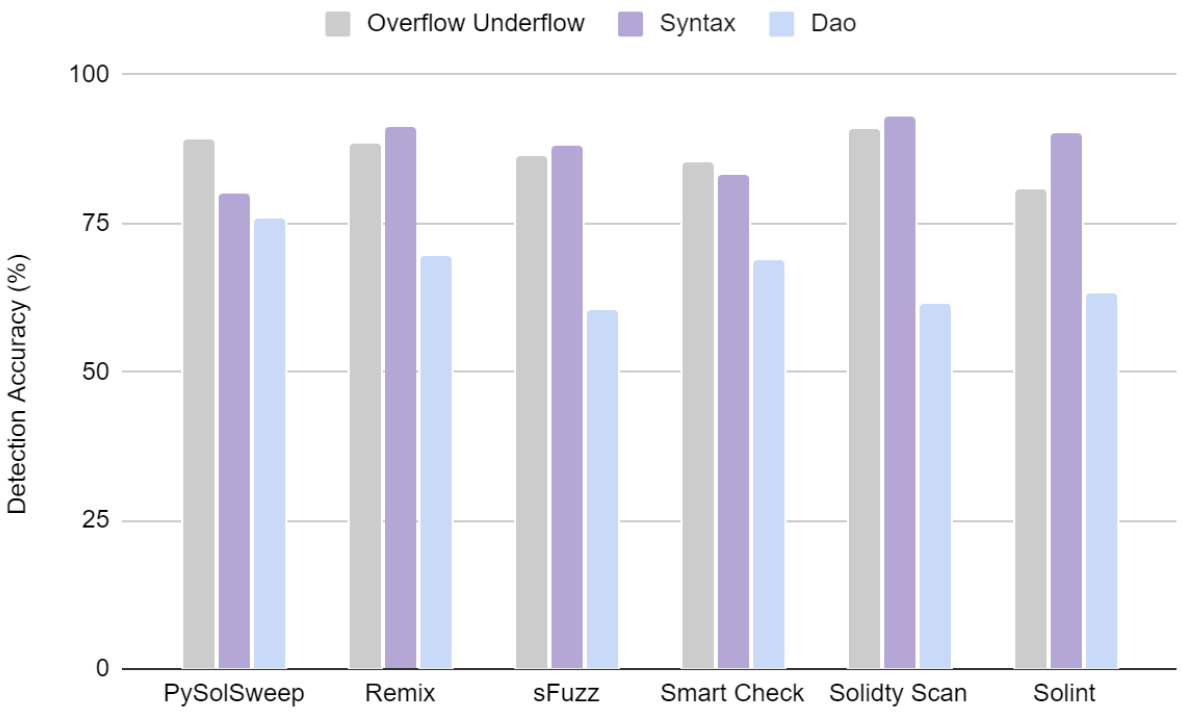


Figure 16: Comparative Accuracy Results of Evaluated Static Analysis Tools

In terms of the aim to achieve equivalent accuracy, the proposed tool deviated 5.2% from the average overall detection accuracy. As illustrated above (**Figure 16**), detection of overflow/underflow bugs was equally matched across all tools with the proposed tool PySolSweep. For syntax bugs, PySolSweep deviated 6% from the average detection accuracy. For DAO bugs, the project's tool PySolSweep was the most accurate, exceeding the average detection accuracy by 12%. This result directly coincided with the hypothesis that the proposed tool PySolSweep would have slightly degraded performance in detection accuracy, due to its implementation not following the common general symbolic execution or ANTLR parsing infrastructure approach to Static Analysis. Nevertheless, the proposed tool demonstrated that it achieved the aim of closely matching the detection accuracy of existing tools. However, it was interesting to observe that for general DAO bugs, the proposed tool exceeded the average detection accuracy of existing Static Analysis tools. This finding can be explained by the increased coverage offered by PySolSweep against the DAO BAT (Bug Attack Theme) of 5 bugs and vulnerabilities compared to that of 1 by

the existing tools. This coverage disparity resulted in a potentially lower accuracy due to a combination of false positives and smaller margin for error in the ratio of correctness with this lower number. The detection accuracy of each tool was calculated using the following formula (**Figure 17**).

$$Accuracy = \frac{Total\ Bugs\ BAT\ Detected - Total\ False\ Positive\ BAT\ Bugs\ Detected}{Total\ Bugs\ Detected}$$

Figure 17: Equation used to determine Static Analysis Tool Accuracy

It was clear that the aim of accuracy was a slight limitation, But the aim of closely matching existing tools' accuracy is achieved by the proposed tool PySolSweep. Furthermore, the accuracy in detecting DAO BAT bugs was exceeded, suggesting the approach of increased DAO attack bug and vulnerability coverage, yielded higher accuracy in detection of DAO bugs and vulnerabilities for Static Analysis tools.

5.1.2 Overflow/Underflow Detection

Table 9: Overflow/Underflow Detection Results

Tool	Total	False Positive	CFD Verified
Solidity Scan	*142*	*13*	*129*
SmartCheck	*370*	*54*	*316*
sFuzz	*1488*	*206*	*1282*
Solint	*175*	*27*	*148*
Remix	*554*	*65*	*489*
*PySolSweep**	*2083*	*224*	*1859*

The first BAT (Bug Attack Theme) included that of bugs and vulnerabilities which could be classed as Overflow/Underflow attack vulnerabilities. As illuminated in **Table 9** above, the project's proposed tool detected the highest number of Overflow/Underflow bugs. It was also the highest when it comes to false-positive detection. This phenomenon could be explained due to the higher volume due of increased coverage. As evident in the proposed coverage (*5.3.5*) both sFuzz and PySolSweep offered significantly more coverage than the other tools. Both these tools were able to detect more than 1000 vulnerabilities and bugs in the Etherscan Smart Contracts. Solidity Scan, SmartCheck and Solint performed the worst in terms of coverage. Their lack of coverage volume can be credited to their lower number of bugs detected for the Overflow/Underflow BAT, which could make a Smart Contract more vulnerable to an Overflow/Underflow exploit attack. Furthermore the proposed tool PySolSweep was able to detect more than 600 CFD verified new bugs and vulnerabilities than the next best performer sFuzz. Hence, demonstrating that one of the project's aims and limitation of increased coverage Overflow/Underflow BAT was achieved.

5.1.3 Syntax Detection

Table 10: Syntax Detection Results

Tool	Total	False Positive	CFD Verified
Solidity Scan	*566*	*39*	*527*
SmartCheck	*1292*	*216*	*1076*
sFuzz	*2254*	*294*	*1960*
Solint	*764*	*74*	*690*
Remix	*1194*	*104*	*1090*
*PySolSweep**	*2626*	*302*	*2324*

The second BAT (Bug Attack Theme) was that of bugs and vulnerabilities which were classified as Syntax BAT. It can be observed (**Table 10**) that again the proposed tool PySolSweep and sFuzz performed above the other tools. Both Remix and SmartCheck performed relatively equally, as expected given their Syntax BAT coverage ranged from 8-10 bugs. The tool sFuzz's gap to the other tools is also explained by its higher Syntax BAT coverage being 14 bugs. Whilst the project's proposed tool integrated 24 bugs and vulnerabilities within the Syntax BAT class, to explain its high CFD verified bug and vulnerability detection number. Both Solint and Solidity Scan were lower than the tools evaluated, more than 1500 bugs and vulnerabilities deviation from the PySolSweep. This observation is also explained by these two tools having a narrow Syntax BAT proposed coverage number of 6-8 bugs and vulnerabilities. Overall, the aim and limitation of increased Syntax BAT coverage was achieved with 400 more bugs and vulnerabilities detected than the best existing Static Analysis tool. Hence, reducing the likelihood of a Syntax exploit for an adversary on a Solidity Smart Contract, compared to existing tools.

5.1.4 DAO General Detection

Table 11: DAO General Detection Results

Tool	Total	False Positive	CFD Verified
Solidity Scan	*0*	*0*	*0*
SmartCheck	*0*	*0*	*0*
sFuzz	*41*	*15*	*26*
Solint	*0*	*0*	*0*
Remix	*0*	*0*	*0*
*PySolSweep**	*324*	*87*	*237*

The final general BAT of DAO yielded the results which can be observed above (**Table 11**). These results in Smart Contracts which did not integrate a withdraw/transfer function illuminated a worrying pattern of current tools failing to consider DAO bugs and vulnerabilities in non withdraw/transfer functional Smart Contracts. Only the project's proposed tool PySolSweep and sFuzz were able to detect any DAO bugs and vulnerabilities. With the number detected by the project's tool an 800% increase over that of its nearest best performer sFuzz. The sFuzz tool was only able to detect one bug of Reentracy conditions. Whilst the proposed tool extended on this to also detect the lack of a precaution counter measure being a modifier function condition to block the user function interactions characteristics that lead to a Reentrancy attack. The other stools Remix, Solint, SmartCheck and Solidity Scan failed to detect a single DAO bug in the general Contracts analysed. Thus, the aim and limitation of increased DAO BAT coverage was satisfied, therefore offering Solidity Smart Contracts increased coverage against a DAO attack with both bugs and countermeasures being checked.

5.1.4 DAO Withdraw Function Detection

Table 12: DAO Withdraw Function Detection Results

Tool	Total	False Positive	CFD Verified
Solidity Scan	*7*	*2*	*5*
SmartCheck	*18*	*6*	*12*
sFuzz	*49*	*13*	*36*
Solint	*67*	*18*	*49*
Remix	*82*	*21*	*61*
*PySolSweep**	*230*	*51*	*179*

The result illustrated above (**Table 12**) exemplifies that of Smart Contracts Statically analysed, of which involve transactions using a withdraw/transfer function. It was observed that the tools Solidity Scan and SmartCheck performed weakest with the lowest number of DAO withdrawal function bugs detected. These bugs detected only included one variant of Reentracy being that the function failed to check through '*require*' that the balance exceeded the withdrawal amount. The tools sFuzz, Solint and Remix performed mediocrely, with the additional variant of Reentrancy being that of checking that '*transfer*' is used instead of '*call.value*' to update balance, which prevents recursive withdrawals. Meanwhile the project's proposed tool PySolSweep performed the best against the other tools, detecting more than 3-fold the average performing tool for DAO withdraw/transfer BAT (Bug Attack Theme). This extended coverage included Contract lock, external call and check-effect-interact checks, accounting for the increased coverage. Hence, this experiment's result satisfied the aim and limitation of increased DAO BAT withdraw function having increased coverage against the DAO attack by detecting more bugs, vulnerabilities and precautions.

5.1.5 CFD Logic Consensus

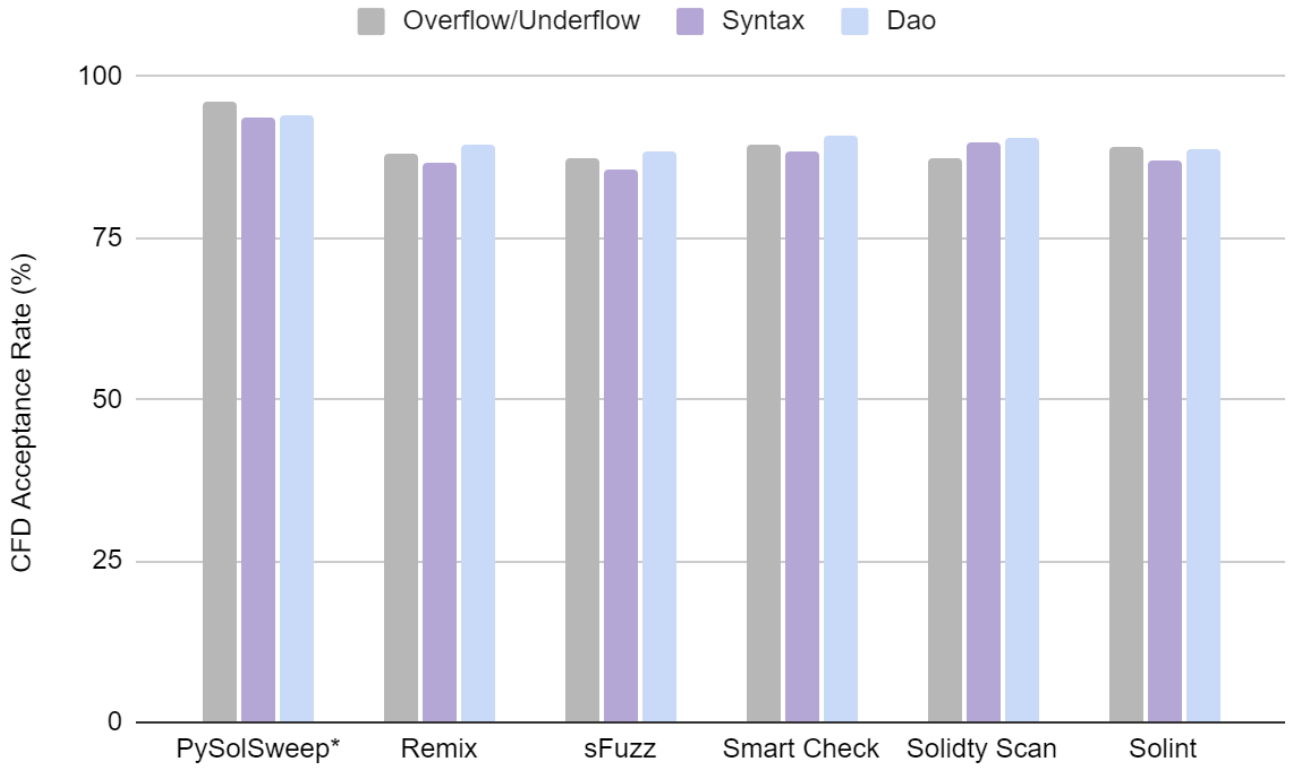


Figure 18: Evaluated Static Analysis Tools CFD Consensus Acceptance Rate Results

The experiment also collected metrics on the general consensus acceptance rates of the Control Flow Diagram (CFD) logic for each of the tools. As evident in (**Figure** 18) above, the project's proposed tool PySolSweep had the highest acceptance rate, with the majority of the three BAT's existing in the 90 percent range. This was expected since the code detection logic for PySolSweep was formed on the basis of the proposed CFD logic. The existing tools acceptance rate for the CFD logic ranged from mid 80s to the 90 percent range. Which was slightly less than the proposed tools CFD logic acceptance rate of a bug or vulnerability being correctly identified. However it can clearly be observed that the CFD path violation logic used to verify that a bug or vulnerability exists within a Static Analysis identification holds credible and strong across all evaluated Static Analysis tools.

BAT-1	BAT-2	BAT-3
89.62	88.51	90.36

Table 13: Average BAT CFD Consensus Rates

This notion of CFD logic credibility is reinforced, as can be observed in **Table 13**. The general consensus of agreement across all tools averaged in the 90 percent range. This finding implied that for the majority of bugs and vulnerabilities detected, the CFD logic was correct in verifying that a true positive condition did exist. However, there was still room observed for improvement.

For Overflow/Underflow (BAT-1) the most common disagreement between all tools and CFD logic was that of type inference. Such that the majority of tools disagreed with the CFD violation path of variables being undeclared.

Whilst for Syntax (BAT-2), general consensus was most commonly not achieved with the Address Zero and transfer bugs and vulnerabilities. With the existing tools suggesting violation paths taken by the CFD logic to be incorrect.

Finally, for DAO (BAT-3) the majority of existing tools failed to agree on the vulnerability of Reentracy to matching the CFD violation path logic. This included the common disagreement of how an update to the balance state variable did indeed create a vulnerable Reentrancy attack condition.

This common lack of consensus suggests a slight limitation of the proposed CFD bug and vulnerability violation paths. However, holistically having a 90 percent consensus agreement between all tools, suggested that the project's proposed CFD logic for bug and vulnerability detection could be verified as credible.

5.6 Results Insights

5.6.1 Overflow/Underflow (BAT-1) Insights

The most Overflow/Underflow BAT (Bug Attack Theme) bugs and vulnerabilities detected by tools was that of Smart Contracts lacking a safety measure Safe Math library. This BAT-1 precautionary library would automatically throw an error handling exception if an arithmetic operation would induce an Overflow/Underflow condition. The performing of arithmetic operations such as '+' without the Safe Math function '*.add*' would not be caught for a potential Overflow/Underflow of the numerical operation. With a minimum of 129 and a maximum of 1859 BAT-1 bugs detected, the majority being linked to this vulnerability was a straddling discovery that Etherscan blockchain deployed Smart Contracts did not coincide with the Safe Math BAT-1 counter measure.

5.6.2 Syntax (BAT-2) Insights

The most common Syntax BAT-2 bug and vulnerability was the defining of the Smart Contract's compiler version as dynamic. The consequence was the assumption that the code would compile with any future version of Solidity. Defining the Smart Contract dynamically using the '^' operand rather than Statically without it could lead to the Smart Contract being inoperable with Solidity compiler updates. The second most common BAT-2 bug and vulnerability observed by only the project's proposed tool PySolSweep was that of excessive Gas usage with function calls in for/while loops, lack of a precaution check if a passed function address variable is not '*0 or 0x0*'. As well as low level calls which many Etherscan Smart Contracts used '*send*' instead of '*transfer*' which would automatically throw an exception if a transaction failed, whereas the former would not have any transaction success notification. This increased coverage of prevalent BAT-2 bugs and vulnerabilities only being detected by the proposed projects tool, reinforces that overcoming limitations of lower BAT-2 coverage by existing Static Analysis tools left Smart Contracts more vulnerable to a Syntax attack.

5.6.3 DAO (BAT-3) Insights

For the final BAT-3, being that of DAO many key takeaways and insights could be extracted from both the general DAO and withdraw/transfer DAO experiments. For the general DAO experiment, only sFuzz and the proposed tool PySolSweep detected any DAO bugs. With the most common bus and vulnerabilities detected only by the proposed tool was that of the Reentrancy condition as well as external call precaution. The medium risk rated external call label was a common vulnerability only detected by the project's tool, this includes the marking of external functions as either trusted/untrusted. If the Etherscan Smart Contracts abided by this proposed solution by labelling the external function, then untrusted inherited balance state variable updates risk would be minimised.

For the DAO withdraw, apart from the common bug detection of a lack of a '*require*' check for the balance exceeding the amount which was detected by all tools. The project's tool highlighted a worrying observation of the majority of Etherscan Smart Contracts failing to correctly implement the check-effect-interact pattern, as a countermeasure against a DAO Reentrancy attack. As well as the majority of Smart Contracts lacked any modifier functions, which prevent value calls that induce Reentrancy for withdraw/transfer functions. This discovery suggested the need for blockchain deployed Smart Contracts to integrate significantly more DAO BAT-3 counter measures as suggested by the project's proposed tool PySolSweep, to reduce the likelihood of a DAO exploit.

5.6.4 General Insights

Reflecting on the work of previous academic papers, it was important to compare the current landscape of Solidity Smart Contract bugs and vulnerabilities as well as Static Analysis tools previously evaluated. The paper titled '*Security Evaluation and Improvement of Solidity Smart Contracts*' found in their study the majority of current Static Analysis tools found bugs and vulnerabilities within the low to medium risk range, with minimal critical bugs and vulnerabilities. As evident in **Figure 19**.

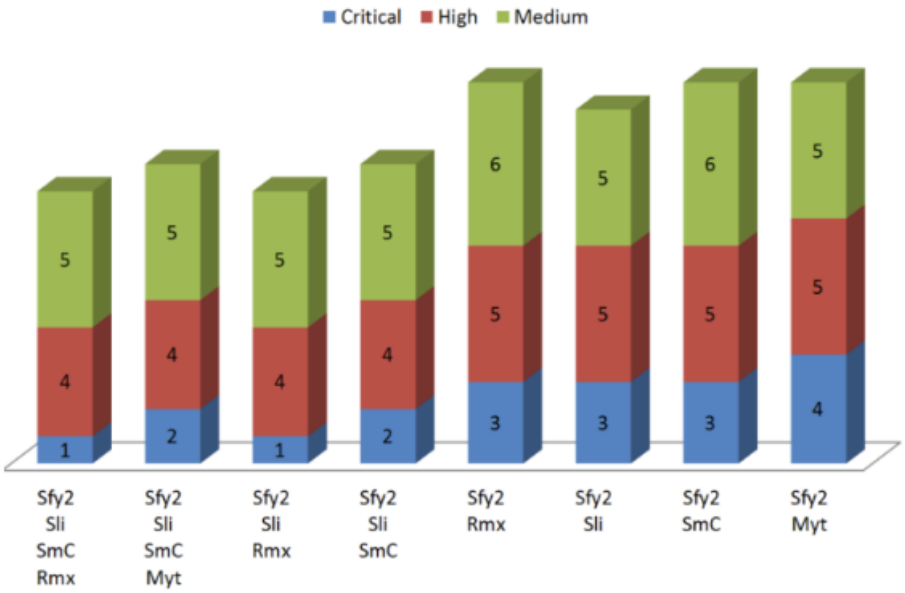


Figure 19: Reviewed Paper's [22] Risk Rating Distribution findings

Having completed testing and evaluation of similar tools on a dataset of Solidity Smart Contracts, the following **Figure 20** illustrates the project's findings of the volume of different impacts of bugs and vulnerabilities by different Static Analysis tools.

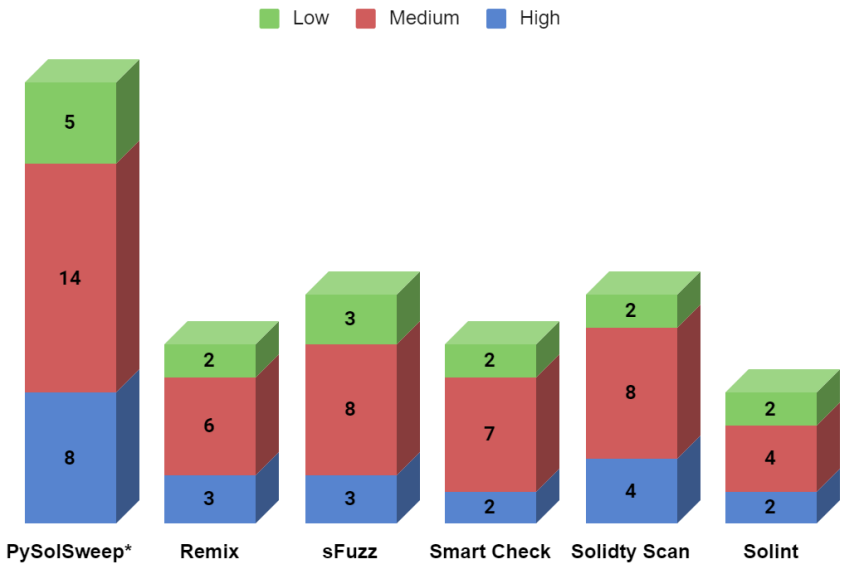


Figure 20: Project's Experiment Evaluation Risk Rating Distribution findings

Comparing the results of previous academic studies [22] evaluation of existing Static Analysis tools on the severity of bugs and vulnerabilities being classified as low, medium or high. It can be observed that with the BAT (Bug Attack Theme) areas of Overflow/Underflow, Syntax and DAO that bugs and vulnerabilities which are most prevalent in deployed Smart Contracts was that of medium impact bugs. This slightly contrasts the academic studies [22] findings of low and medium impact bugs and vulnerabilities holding dominance over high impact. This notion suggested the newer exploits within the 3 BAT exist within current Smart Contracts with medium risk consequences. Another contrast was that of the project's proposed tool, in which the high-risk bugs and vulnerabilities of 8 variants, exceeded that of any bug risk category of other Static Analysis tools. This further exemplified the achievement of overcoming the existing limitation of poor BAT coverage, as well as the current landscape of blockchain deployed Smart Contracts being vulnerable to Overflow/Underflow, Syntax or DAO BAT's with medium to high-risk consequences.

5.6.5 Alternative Approach

Finally, the limitations and shortcomings which arose in conducting this Static Analysis comparative study, include that of the scope of the project being limiting the amount of test Smart Contracts which could be used. With existing papers [14] using a dataset of 7000 Smart Contracts, compared to this projects 150. This alternative approach to increase to a larger test dataset with future work add credibility to the findings of this study, as well as more strongly reinforce conclusions drawn.

The other limitation incorporated that of tools available to compare to the proposed tools. Multiple other Static Analysis tools existed such as MythX, which incur a $250 fee per scan for usage. With an unrestricted budget and increased time frame and personnel resources, conducting this evaluation with commercial Static Analysis tools could have yielded a different result, with their focus more on low to medium impact bugs and vulnerabilities.

A recommendation to improve the credibility and precision of the conducted experiment, in evaluating the projects proposed tool and Control Flow Diagram (CFD) logic would incorporate the following methodology:

- Dataset - 4000 general Smart Contracts and 1000 withdraw/transfer function Smart Contracts
- Expand to 10 Solidity Smart Contract Static Analysis tools to compare to projects tool that is either free or commercially available
- Increase Bug Attack Theme (BAT) classes to other attacks including Parity Sig Multi wallet, Operational and Functional attacks
- Measure more metrics during evaluation, including bug detection volume, false positive rate, false negative rate, CFD bug specific consensus between tools and Analysis time
- Manually peer-review Smart Contract dataset for bugs and vulnerabilities for a control measure number of bugs and vulnerabilities against other tools

Integrating this methodology as an alternative approach should reinforce findings from this project conducted, as well as extend insights into other BAT's and commercial Solidity Static Analysis tools ability to offer adequate coverage against attacks. Furthermore, manually peer-reviewing Smart Contracts for bugs and vulnerabilities, would also allow for false-negative rates to be measured. This approach could offer an insight into an overall BAT coverage Analysis of the current state of Smart Contracts and could further verify the correctness of the project's proposed CFD's violation path logic.

6.0 Conclusion & Future Work

6.1 Project Aim

The aim of this project was to design and develop a Solidity Static Analysis tool, which could be able to overcome the gaps and limitations of existing Static Analysis tools, by constructing Control Flow Diagrams (CFD). A set of 35 CFD's were constructed representing multiple violation logic paths for existing and new bugs and vulnerabilities, which span across three Bug Attack Themes (BAT) of Overflow/Underflow, Syntax and DAO. A GUI Python based Solidity Static Analysis tool '*PySolSweep'* was produced and evaluated, to successfully overcome the existing limitations of current tools of a lack of bug and vulnerability coverage for these 3 BAT's. The credibility and accuracy of both CFD and the proposed Static Analysis tool were validated in an experiment comparing existing tools with Ethereum Blockchain deployed Smart Contracts. The initial plan to produce a CFD set for the 3 BAT's, GUI Solidity Static Analysis tool was produced and evaluated, benchmark criteria generated for a comparative experiment and general insights deduced was evidently satisfied, with the accuracy of the project's tool being slightly below expectations, a slight caveat. Nevertheless, the aims and goals within the scope of this project were achieved holding a significant contribution. With the increased Bug Attack Theme (BAT) coverage approach of Solidity Smart Contracts, demonstrating a better performance against existing tools of securing the safety of these Solidity Smart Contracts against exploits.

6.2 Summary

From reviewing previous academic papers within the space of Solidity Static Program Analysis, this highlighted a range of bugs and vulnerabilities affecting Smart Contracts. With several types of attacks including Overflow/Underflow, Syntax, DAO and Parity Multi-Sig Wallet BAT's (Bug Attack Themes). These papers also highlighted the limitations with current tools available for Static Analysis including their approach of random bug detection, which lacked sufficient coverage against these BAT's. These papers also introduced conditions and behaviours of new bugs and vulnerabilities which fit under these BAT's but did not have any corresponding logical tree infrastructure for detection. Reviewing existing tools it was also evident that suggestive solutions for resolving a bug or vulnerability in code was not available to aid the UX of Solidity Static Analysis. The final limitation observed from papers reviewed was the usability of current

tools. With a lack of cross-platform usage, the majority of tools required substantial program dependencies and a Linux operating system to be installed. As well as their interface being command line based, and the Static Analysis result not being readily available in a file, rather a command line output. These limitations and gaps, alongside background knowledge of bugs and vulnerabilities within three Bug Attack Themes (BAT) of Overflow/Underflow, Syntax and DAO. Formed the scope of this project to develop and evaluate Control Flow Diagrams and a Python based Solidity Static Analysis tool, to bridge this gap and pave the way for future work on this approach of (BAT) coverage of Smart Contracts.

The limitation of a lack of logical tree infrastructure for detection for newer bugs and vulnerabilities within the three BAT's, was addressed with 35 Control Flow Diagrams (CFD's) produced. These CFD's both formed logic for new bugs and vulnerabilities, as well as expanded on existing logical trees with additional logic paths for violations of bugs, vulnerabilities or precautionary counter measures. These logical CFD tree's forks and modules formed the basis for the execution of steps taken for bug and vulnerability detection by the project's proposed Python Static Analysis tool.

The project proposed a Python Based Static Analysis tool, namely '*PySolSweep*', which provided a total coverage of 35 bugs, vulnerabilities and countermeasures Static Analysis by implementing the CFD logic. This tool overcame the existing tools limitation of poor BAT coverage, cross-platform compatibility, a GUI user interface, suggestive bug solution as well as a log file to access the result of the Static Analysis.

The credibility of both CFD logic and a proposed tool '*PySolSweep*', was validated in a competitive experiment. The result of this evaluation supported the logic from the CFD across multiple existing Static Analysis tools. Furthermore, the positive disparity in coverage between existing tools and the projects proposed tool, demonstrated its successful approach of BAT coverage yielded than existing tools random collection for bug coverage against three core BAT's of Overflow/Underflow, Syntax and DAO.

6.3 Significance

The significance and contributions of this project on the Static Analysis of Solidity Smart Contracts was immense. The first major contribution was that of the Control Flow Diagrams (CFD's) constructed. These CFD's extended the logic of 11 existing parse tree infrastructure, as well as proposed 23 new CFD logic for parse tree infrastructure. The evaluation and testing of this CFD logic through a group consensus with multiple other Static tools, validated the credibility of its violation path logic for usage as parse trees in bug and vulnerability detection. Other significant contributions include that of the proposed approach of bug, vulnerability and countermeasures detected being based on Bug Attack Theme (BAT) coverage. The project demonstrated this approach towards program Analysis of determining which bugs and vulnerabilities induce which attack theme, then detect all bugs, vulnerabilities and countermeasures within that attack theme. The experiment evaluation conducted supported this notion, as well as unearthing multiple general insights of existing Solidity Static Analysis tools and Ethereum blockchain deployed Smart Contractors. Contributions from the evaluation included the lack of coverage existing Static Analysis tools provide against core attacks such as Overflow/Underflow, Syntax and DAO. Illustrating the impact of new bugs and vulnerabilities discovered in academic papers [13, 14, 15, 17, 18], and the lack of detection in tools which were released prior to these new bugs and vulnerabilities. The evaluation moreover illuminated a worrying insight into the large volume of medium to high-risk impact bugs and vulnerabilities present in Ethereum blockchain deployed Smart Contracts. Thus, the significance and contributions of this project furthered the knowledge and research in the field of Static Analysis on Solidity Smart Contracts.

6.4 Future Work

Having completed this project on the research Static Analysis of Solidity Smart Contracts, and proposal of Control Flow Diagram (CFD) logic for bugs and vulnerabilities, as well as a Python based tool to overcome deduced limitations and gaps of existing tools. It was evident that multiple limitations and avenues for future work arose from this project. This incorporated that of constructing the Static Analysis tool, using the same CFD logic integrated into a parsing software infrastructure to translate this produced CFD logic into an XML Abstract Syntax Tree (AST). Then run Static Analysis using an XML analyser, such as XPath. This future work would have the potential to improve on the relative mediocre accuracy produced by this project. Hence creating a

high benchmark for other tools to achieve with regards to coverage of bugs and vulnerabilities against specific attacks.

The scope of this project did not include other Bug Attack Themes (BAT's) including that of Parity-Sig Multi Wallet, Operational and Functional attacks. Further research into these BAT's, and what bugs and vulnerabilities can be linked to them, opens further tasks to construct CFD or parse tree logic for detection and integration into a Static Analysis tool. As achieving significant BAT coverage, will reduce the likelihood of a Smart Contract being exploited.

Another potential area for further work included that of a more rigorous evaluation of current Solidity Static Analysis tools. Conducting a similar experiment (*5.0*) with increased test dataset, increased available tools with a paid licence and peer-reviewed dataset Smart Contracts to measure false-negatives detections more accurately. The outcomes of a more rigorous comparative experiment could either reinforce or offer new insights into the current landscape of Solidity Static Analysis tools accuracy, coverage and their weaknesses.

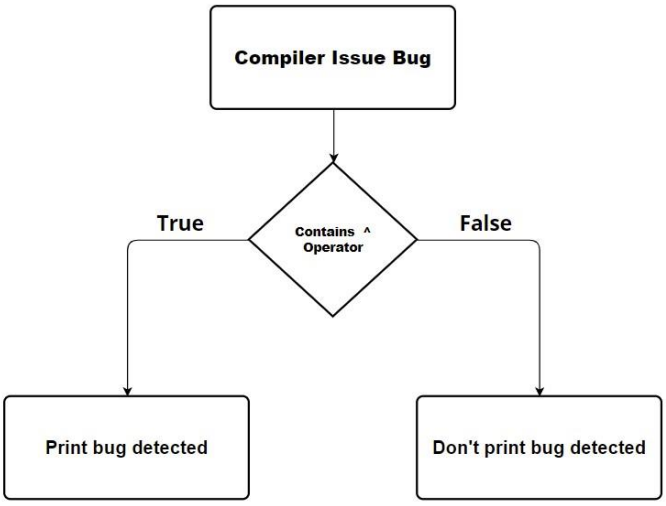
The final area for future work includes dynamic Analysis on Solidity Smart Contracts. Given that the project explored 3 BAT's of Overflow/Underflow, Syntax and DAO. These BAT's can have bugs and vulnerabilities which are out of scope of Static Analysis, whilst within the scope of dynamic Analysis. Hence, future research into which bugs and vulnerabilities are correct with the projects BAT's whilst the Smart Contract is deployed on a testnet Ethereum Blockchain. As well as CFD or parse tree logic to implement in a Solidity Dynamic Analysis tool, allowing for extended coverage of Overflow/Underflow, Syntax and DAO BAT's.

Overall, the results from this study and research were promising contributions to the area of Static Program Analysis of Solidity Smart Contracts. Further work proposed could provide more valuable insights as well reinforce and extend Program Analysis of Solidity Smart Contracts.

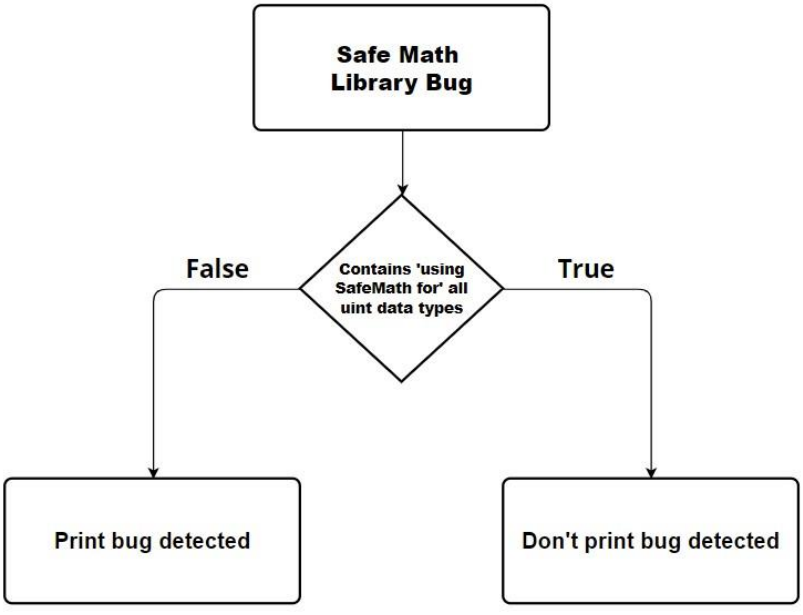
7.0 Appendices

7.1 Appendix A – Control Flow Diagrams

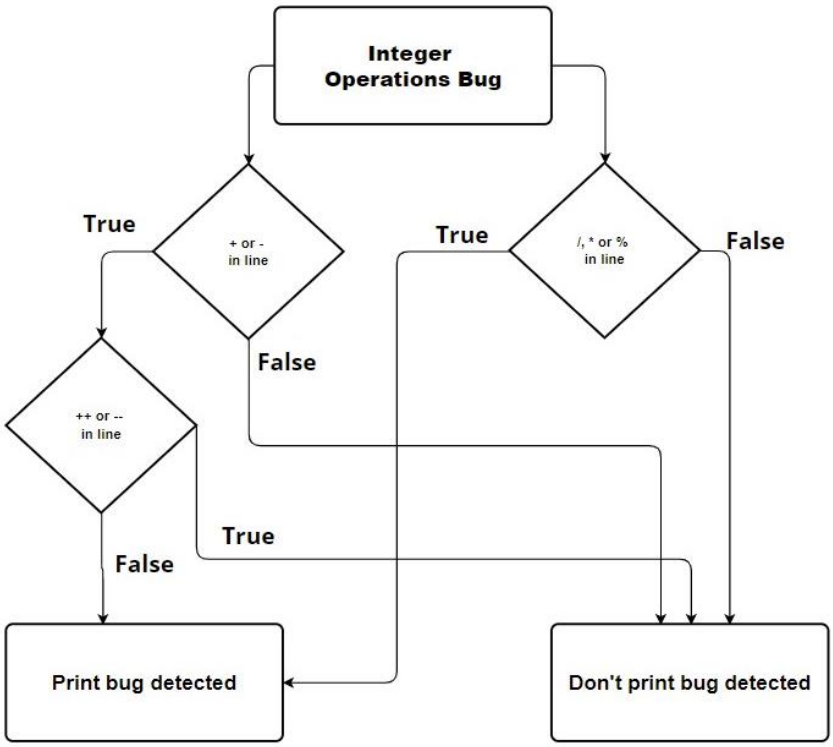
7.1.1 Compiler Issue CFD



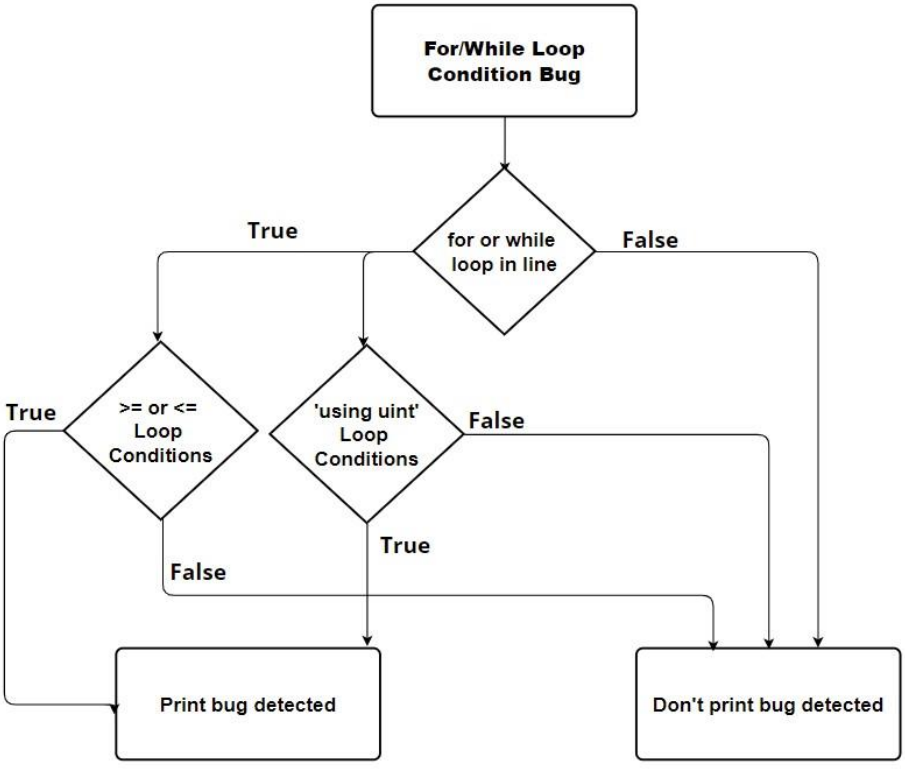
7.1.2 Safe Math CFD



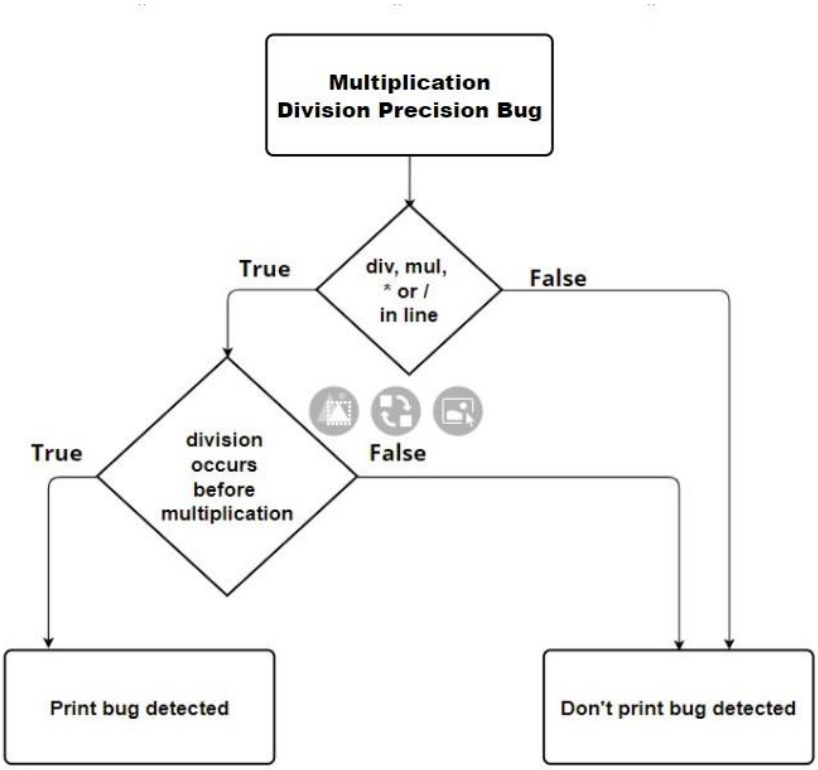
7.1.3 Integer Operations CFD



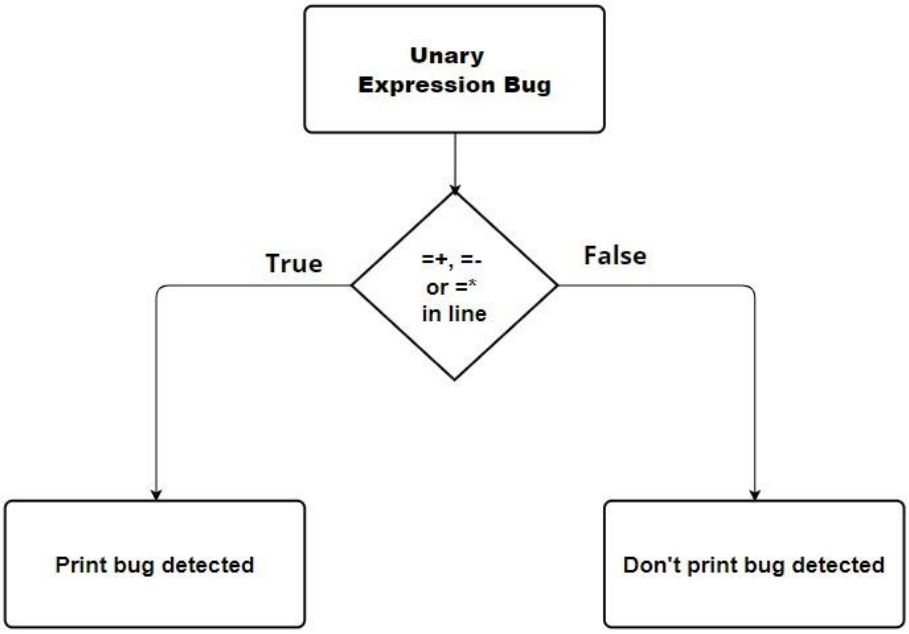
7.1.4 Loop Condition CFD



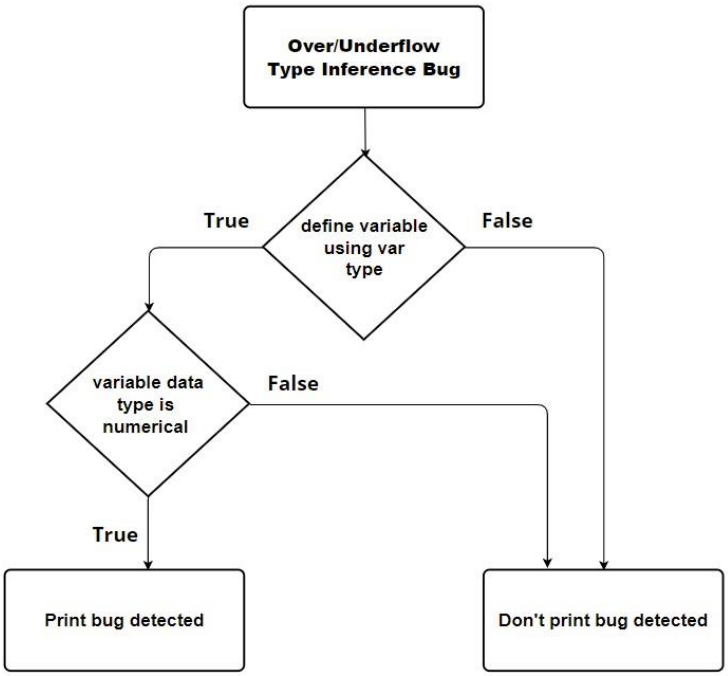
7.1.5 Division Before Multiplication CFD



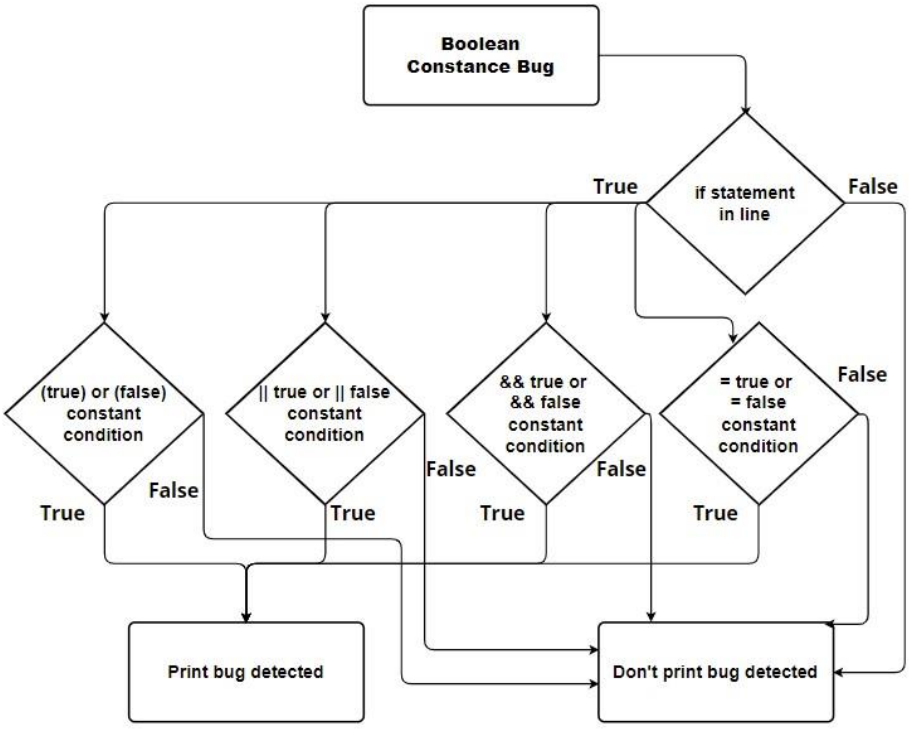
7.1.6 Unary Operations CFD



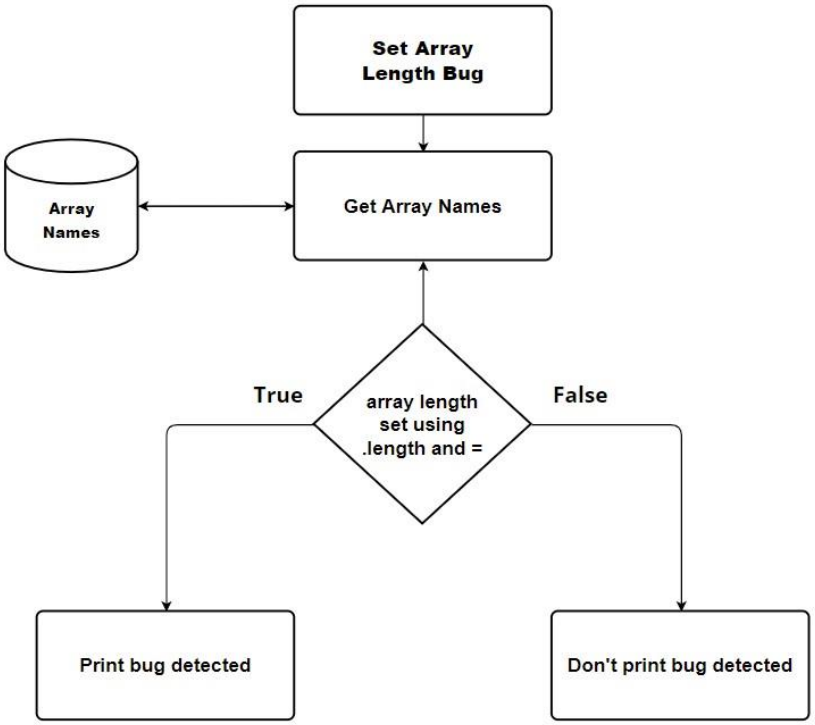
7.1.7 Type Inference CFD



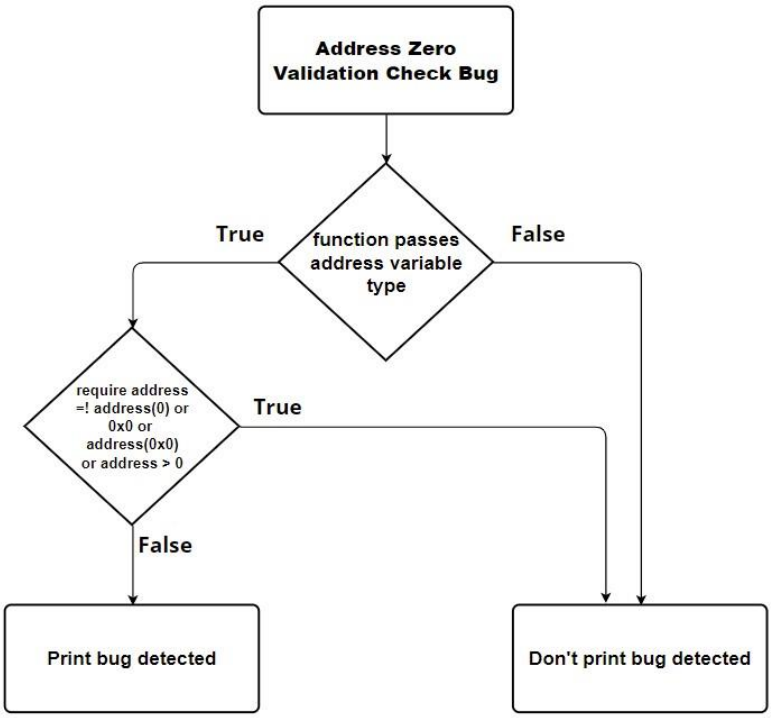
7.1.8 Boolean Constance CFD



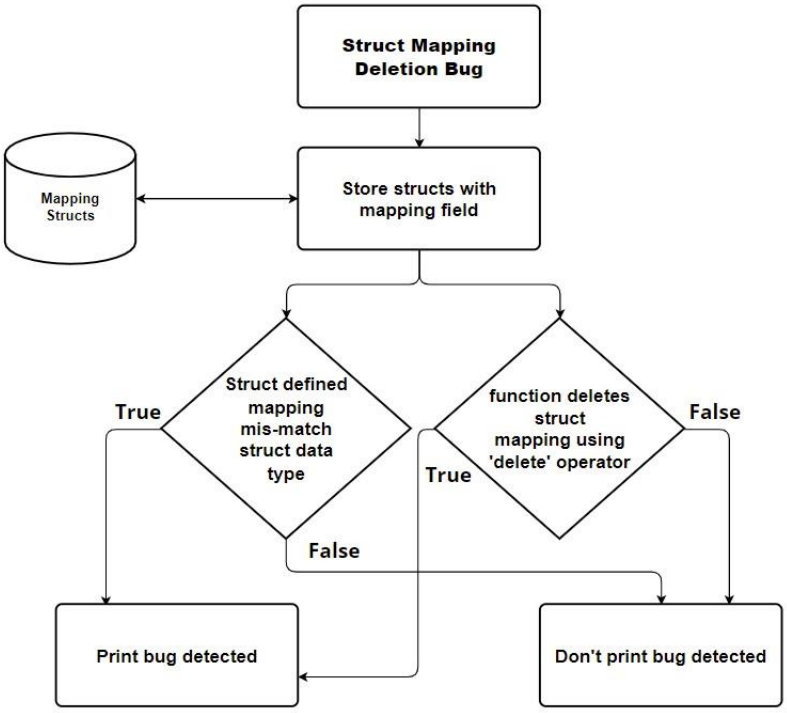
7.1.9 Array Length CFD



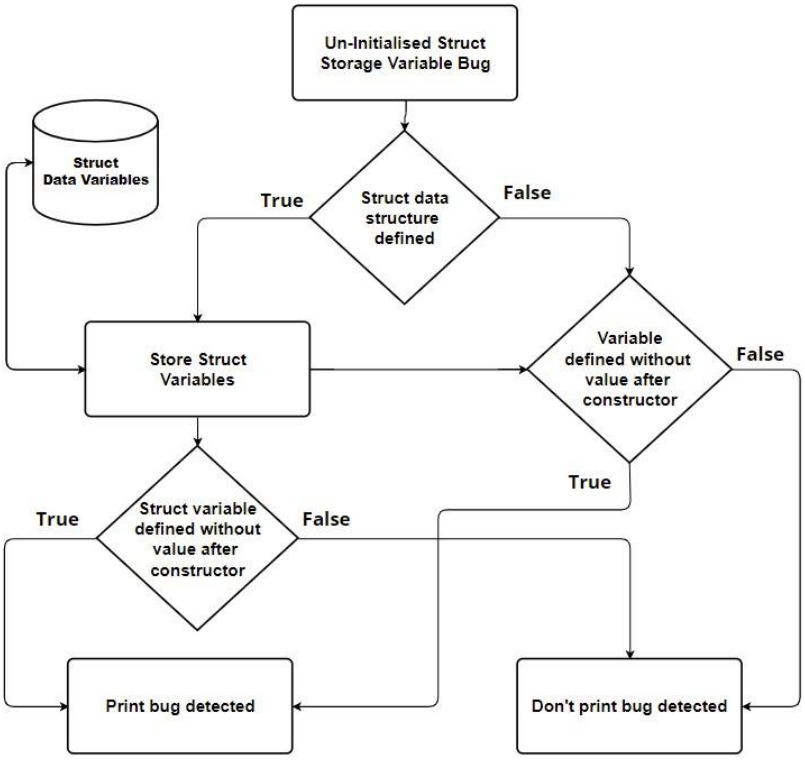
7.1.10 Address Zero CFD



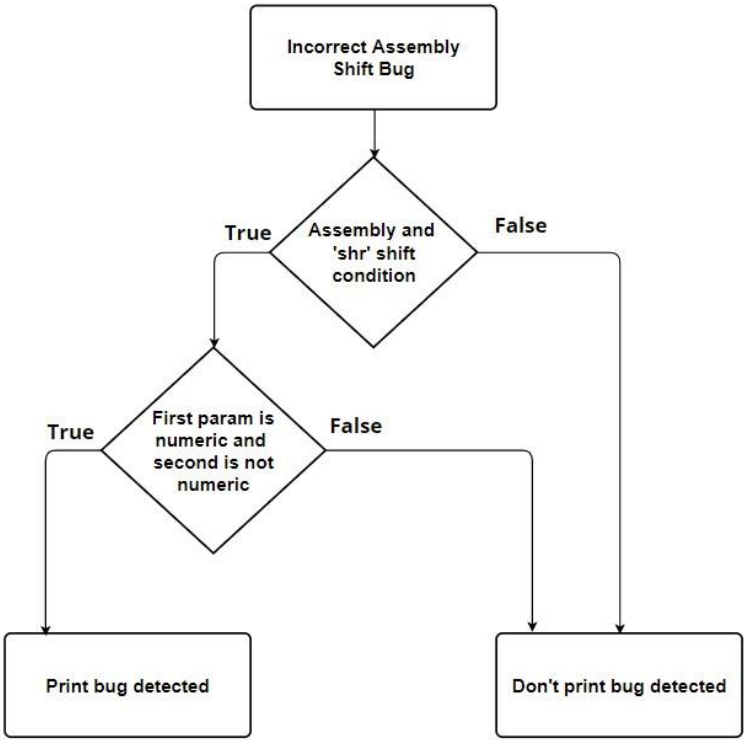
7.1.11 Map Struct Delete CFD



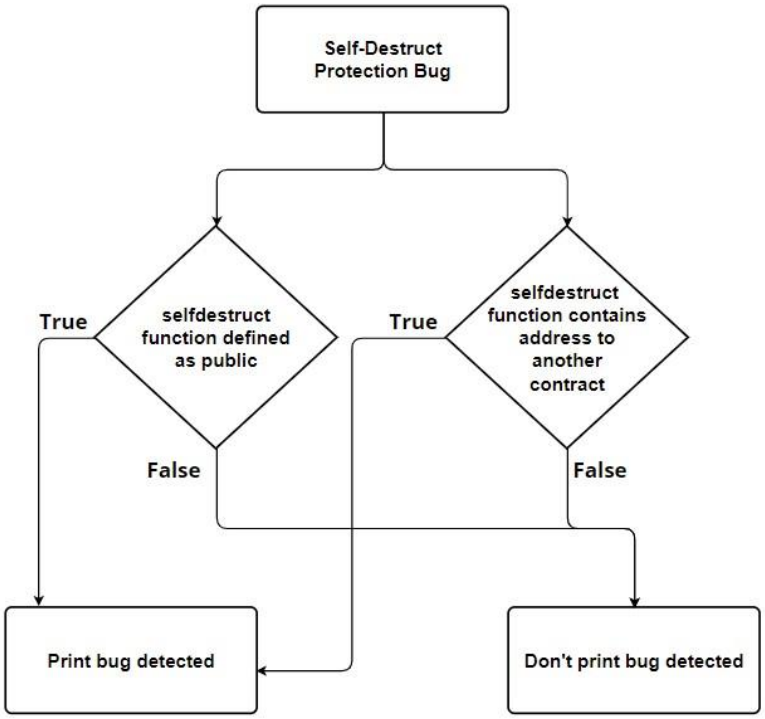
7.1.12 Initialise Storage CFD



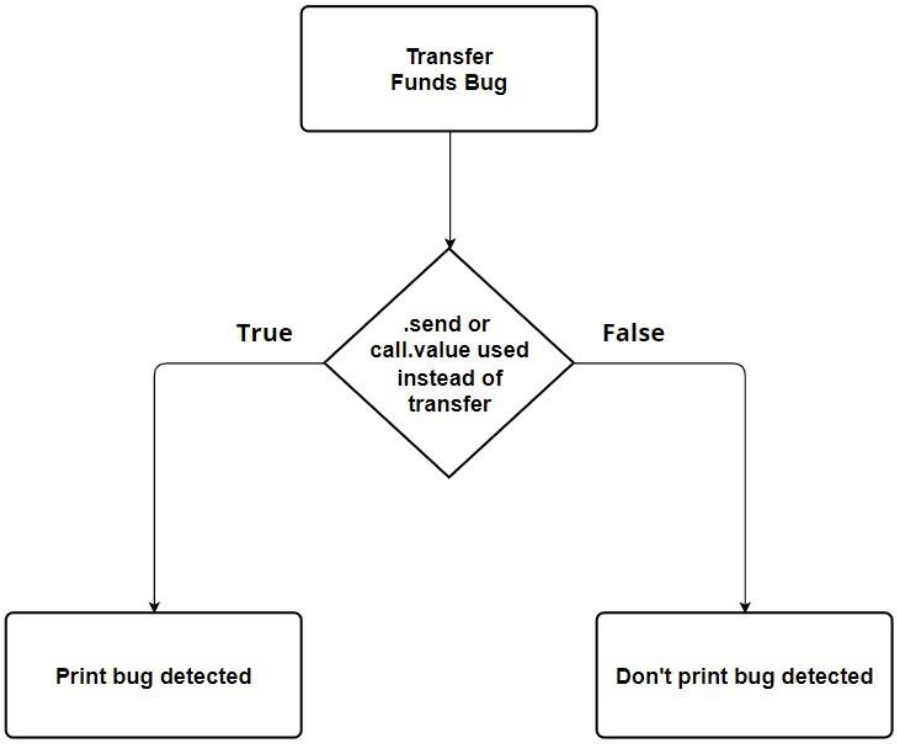
7.1.13 Assembly Shift CFD



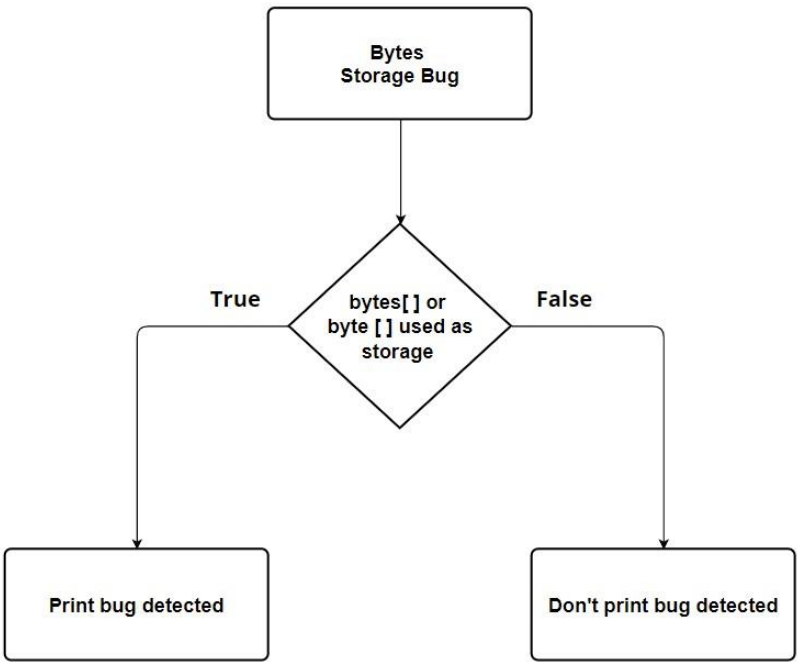
7.1.14 Self Destruct CFD



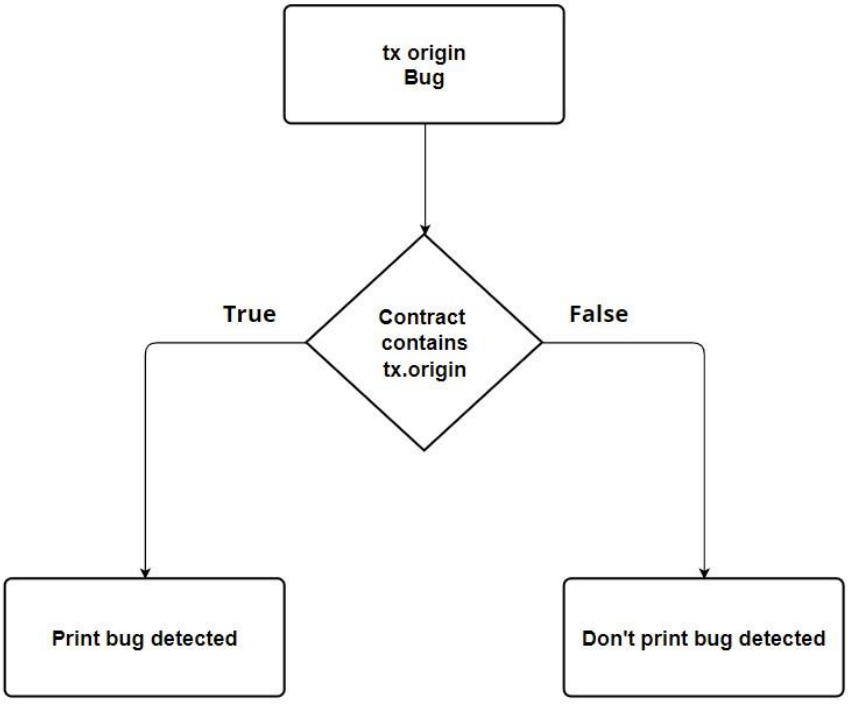
7.1.15 Transfer Condition CFD



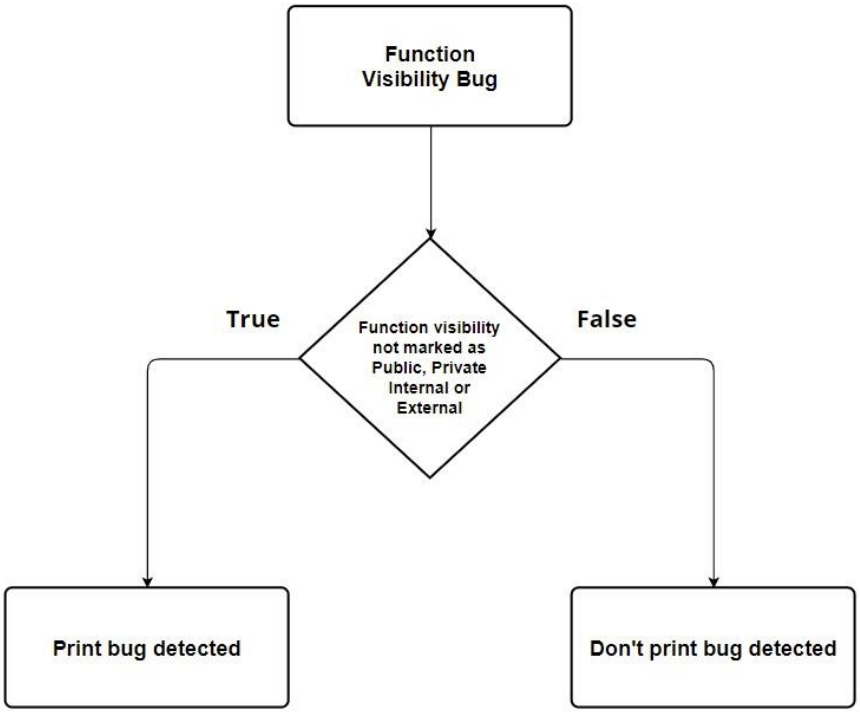
7.1.16 Bytes Condition CFD



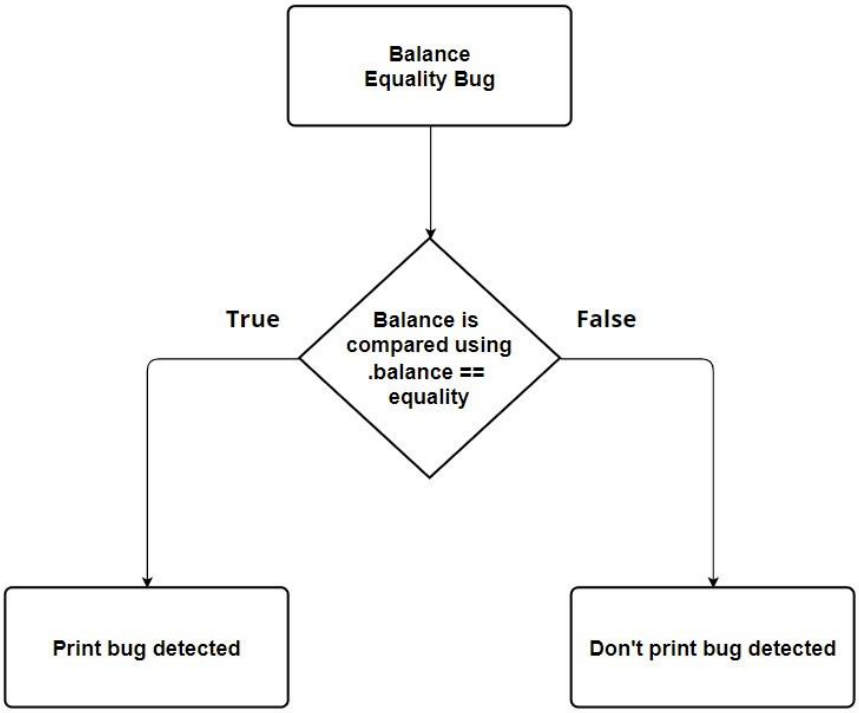
7.1.17 Tx Origin CFD



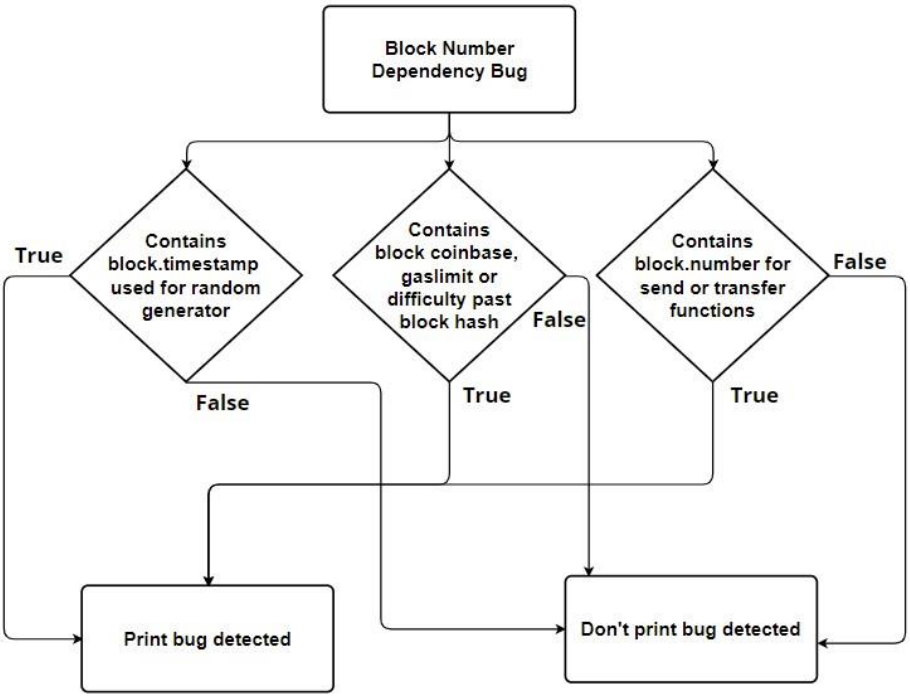
7.1.18 Function Visibility CFD



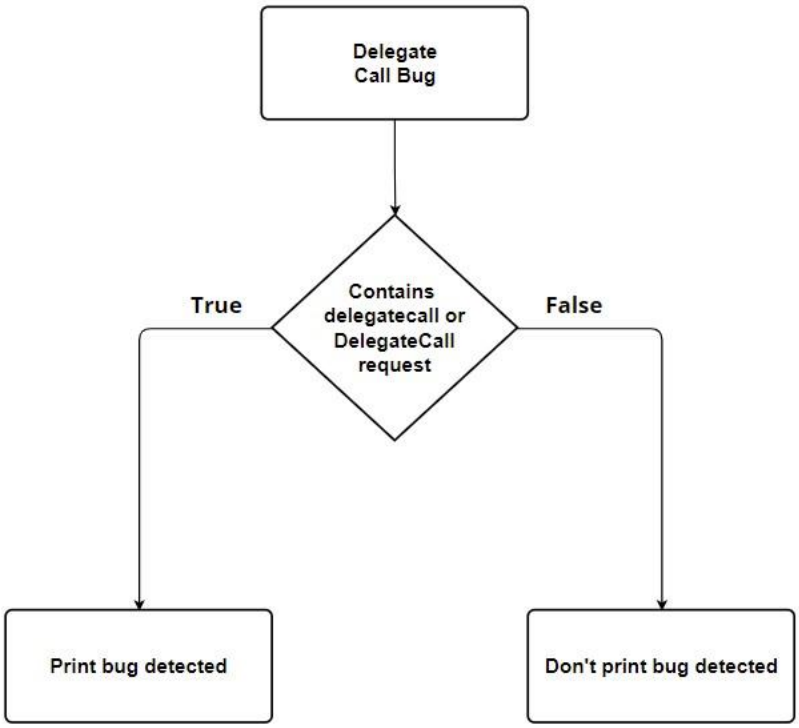
7.1.19 Balance Equality CFD



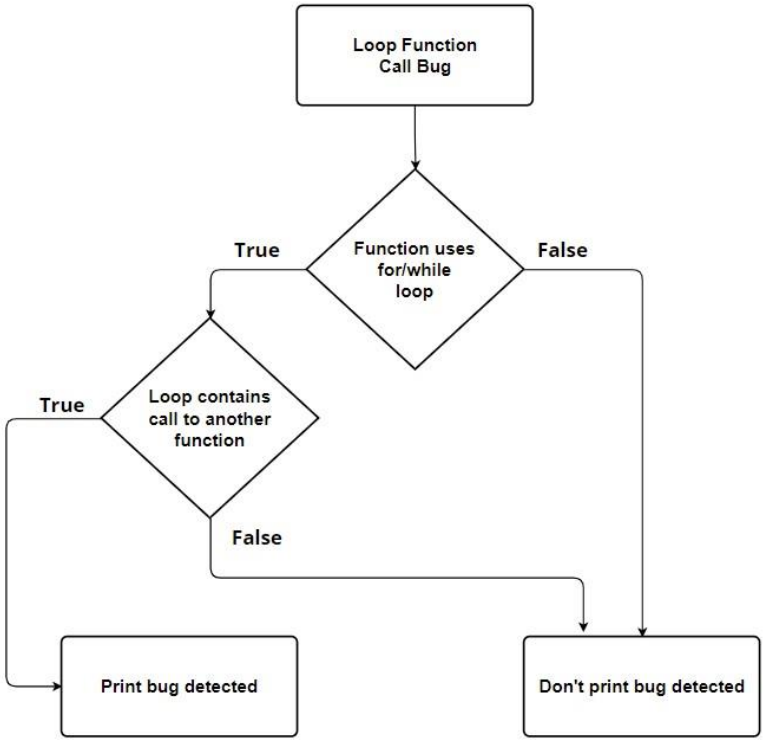
7.1.20 Block Number Dependency CFD



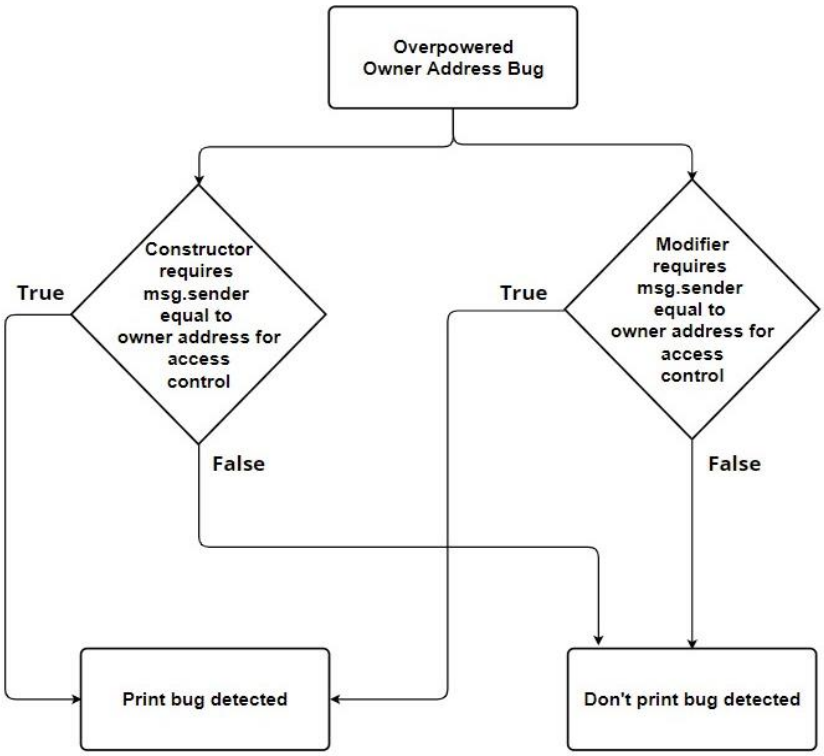
7.1.21 Delegate Call CFD



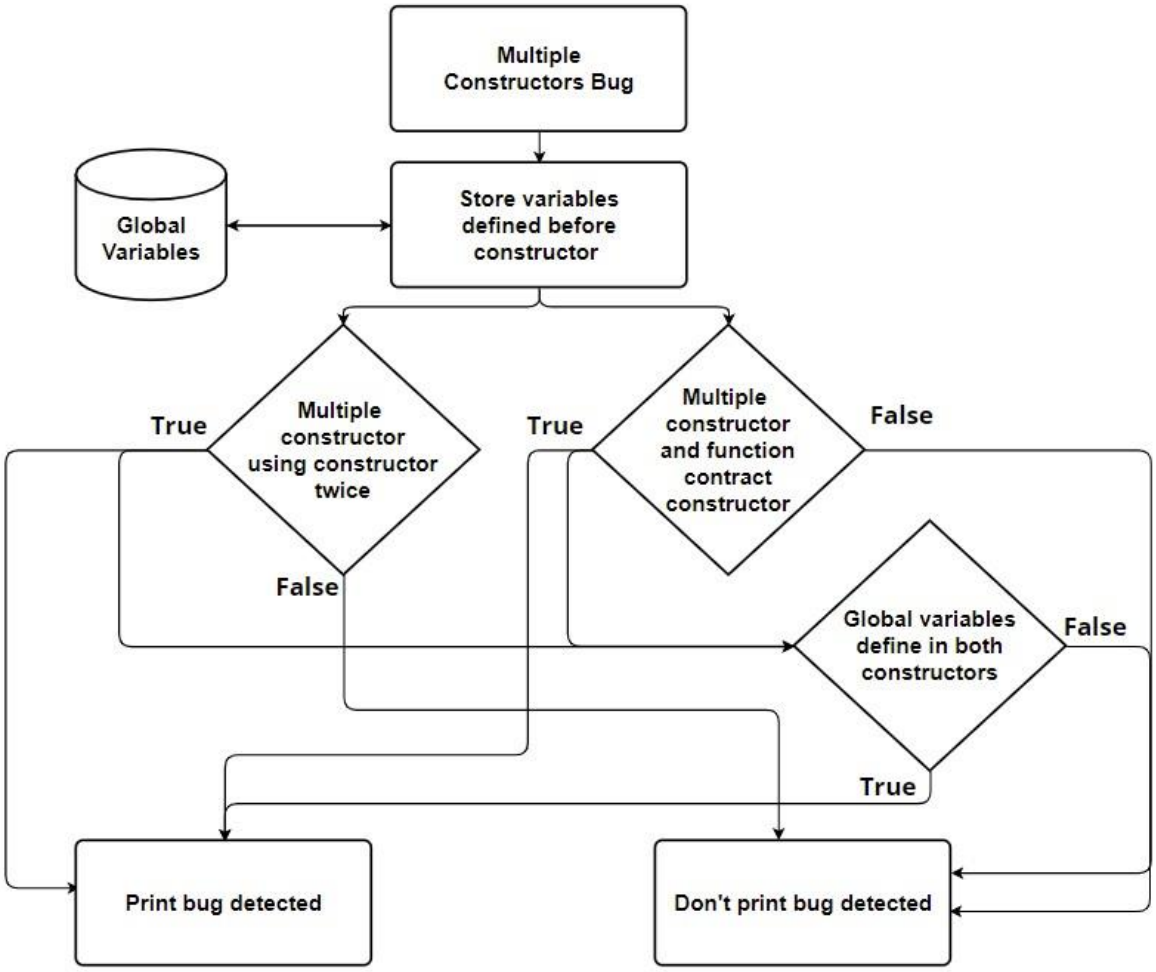
7.1.22 Loop Function Condition CFD



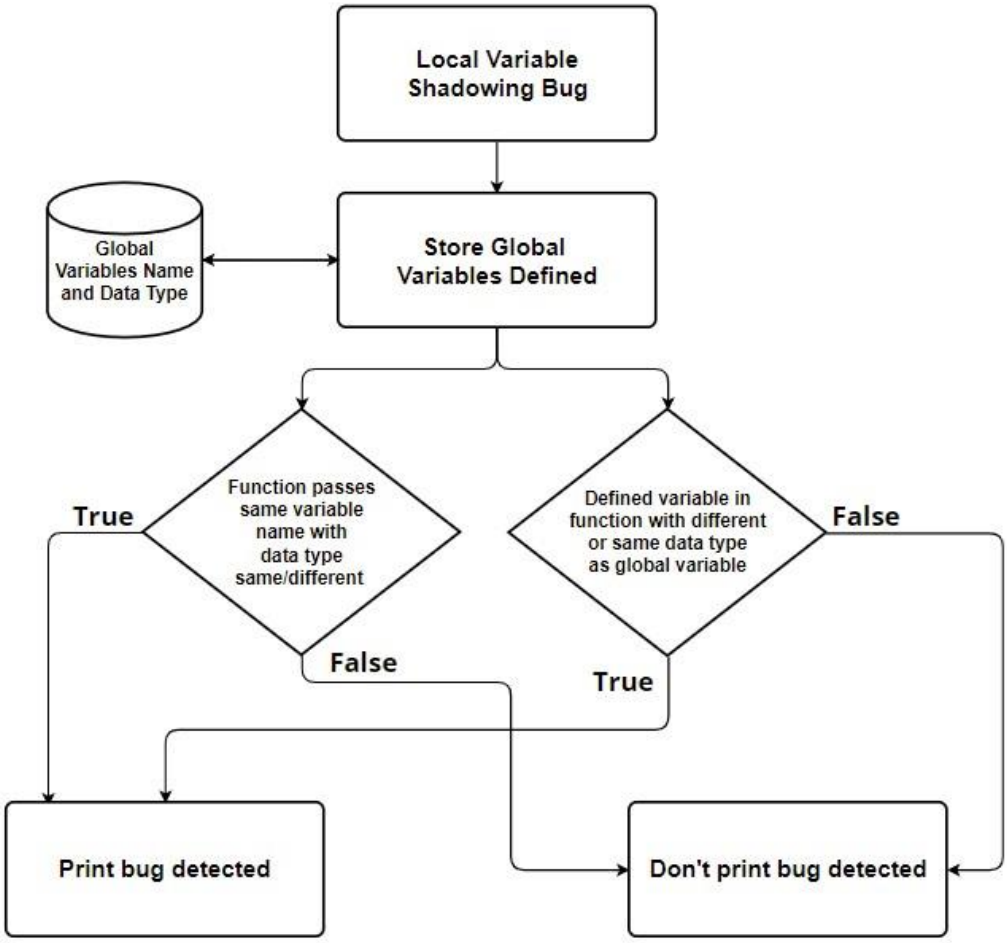
7.1.23 Overpowered Owner CFD



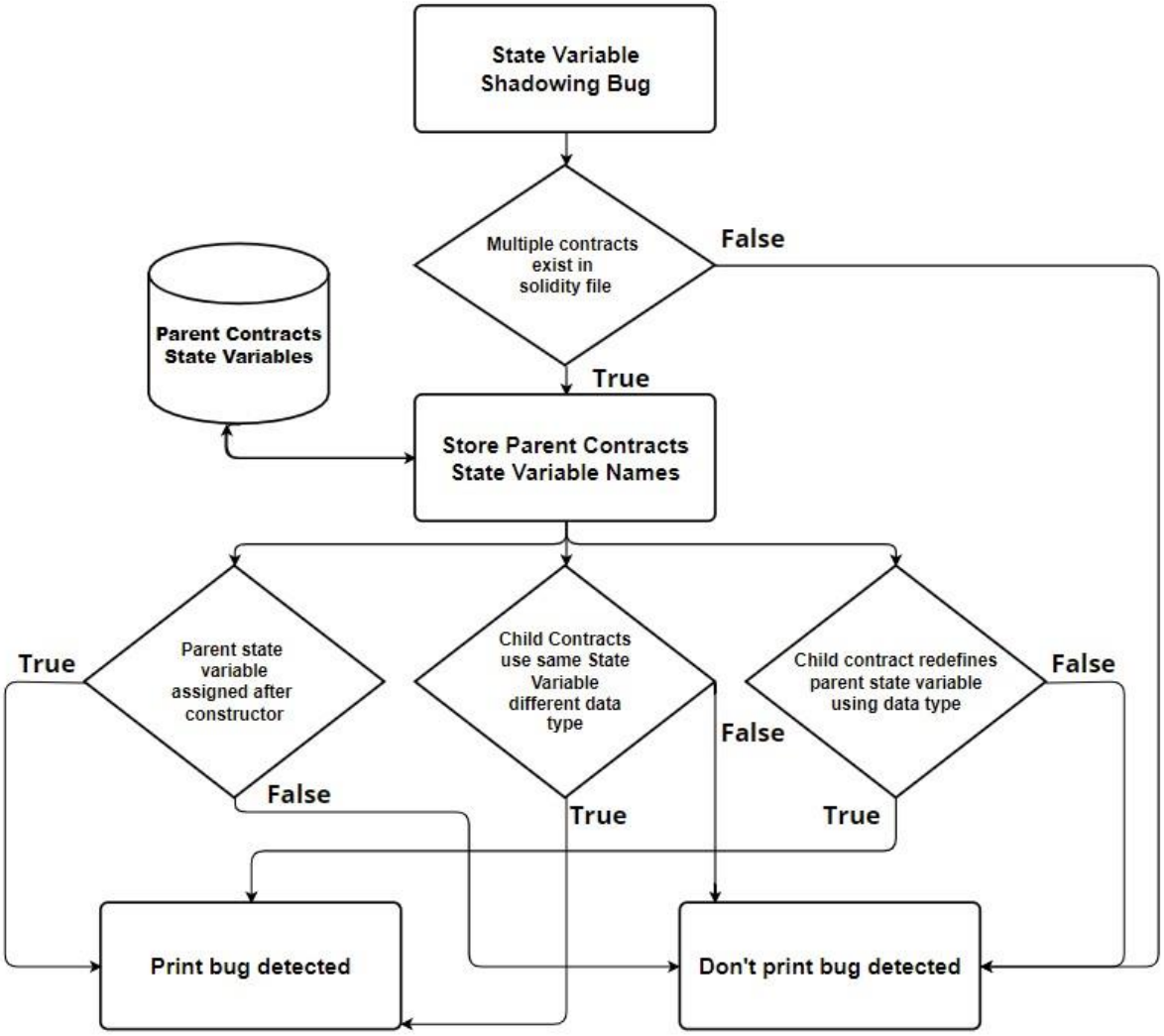
7.1.24 Constructor Initialisation CFD



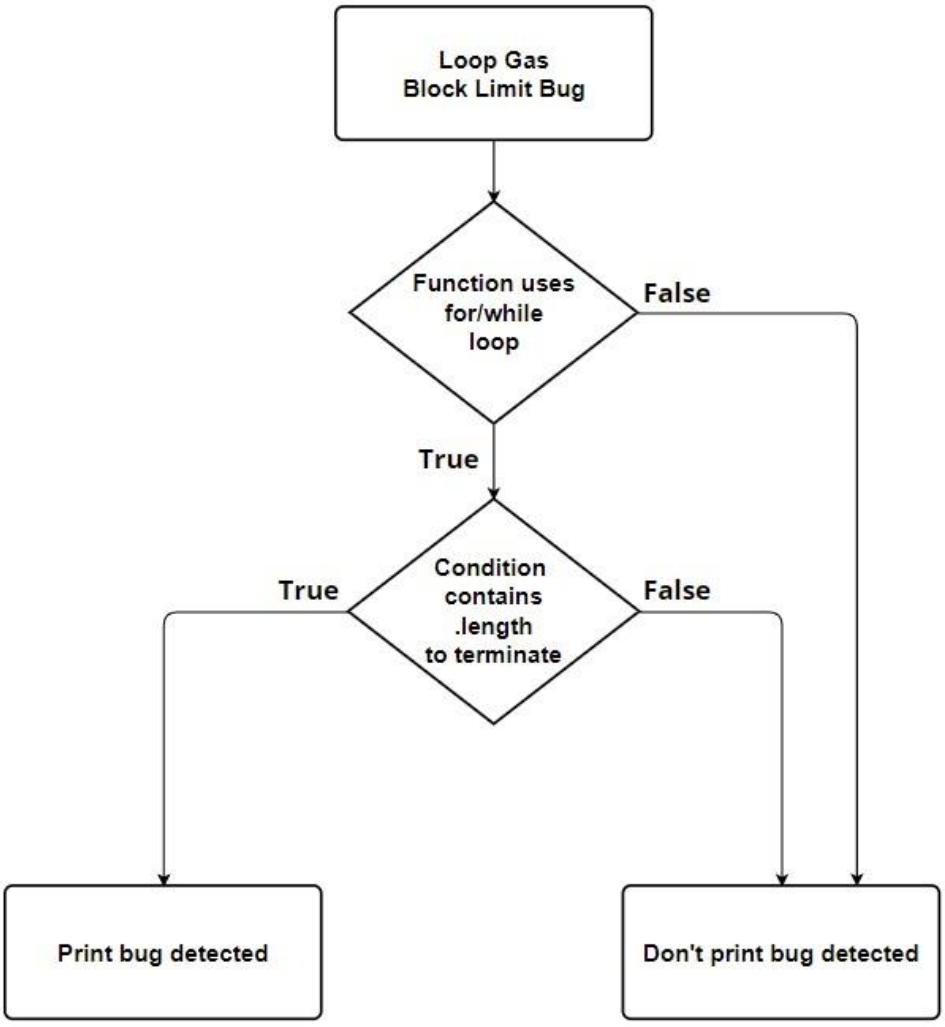
7.1.25 Local Variable Shadowing CFD



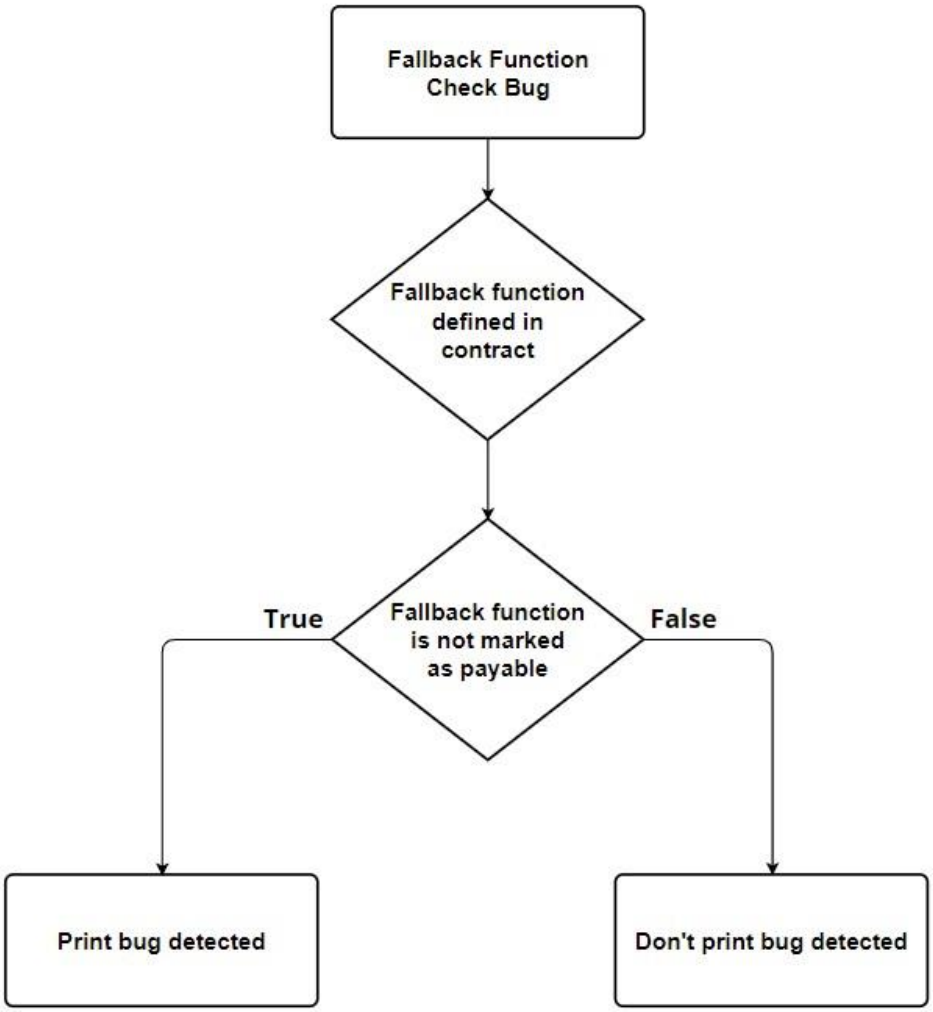
7.1.26 State Variable Shadowing CFD



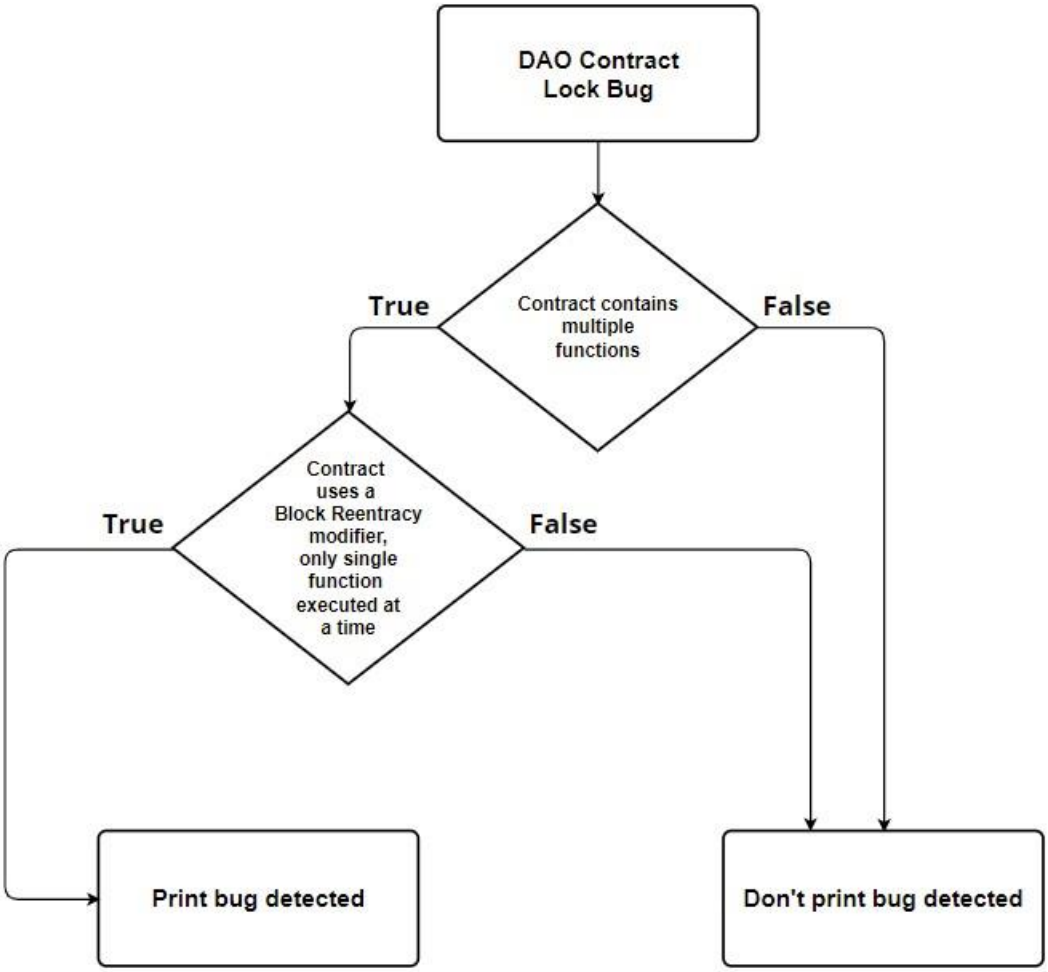
7.1.27 Block Gas Condition CFD



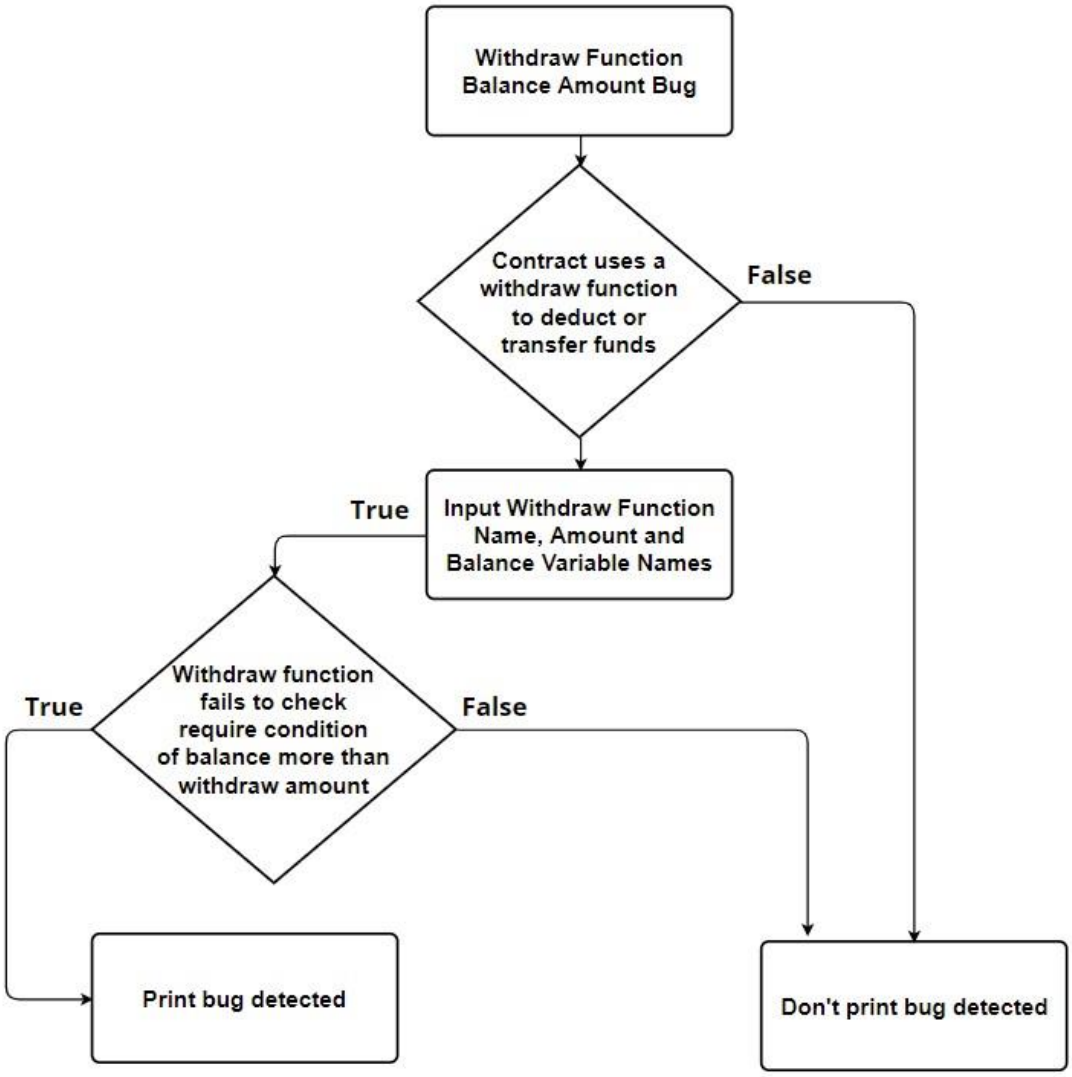
7.1.28 Fallback Function Condition CFD



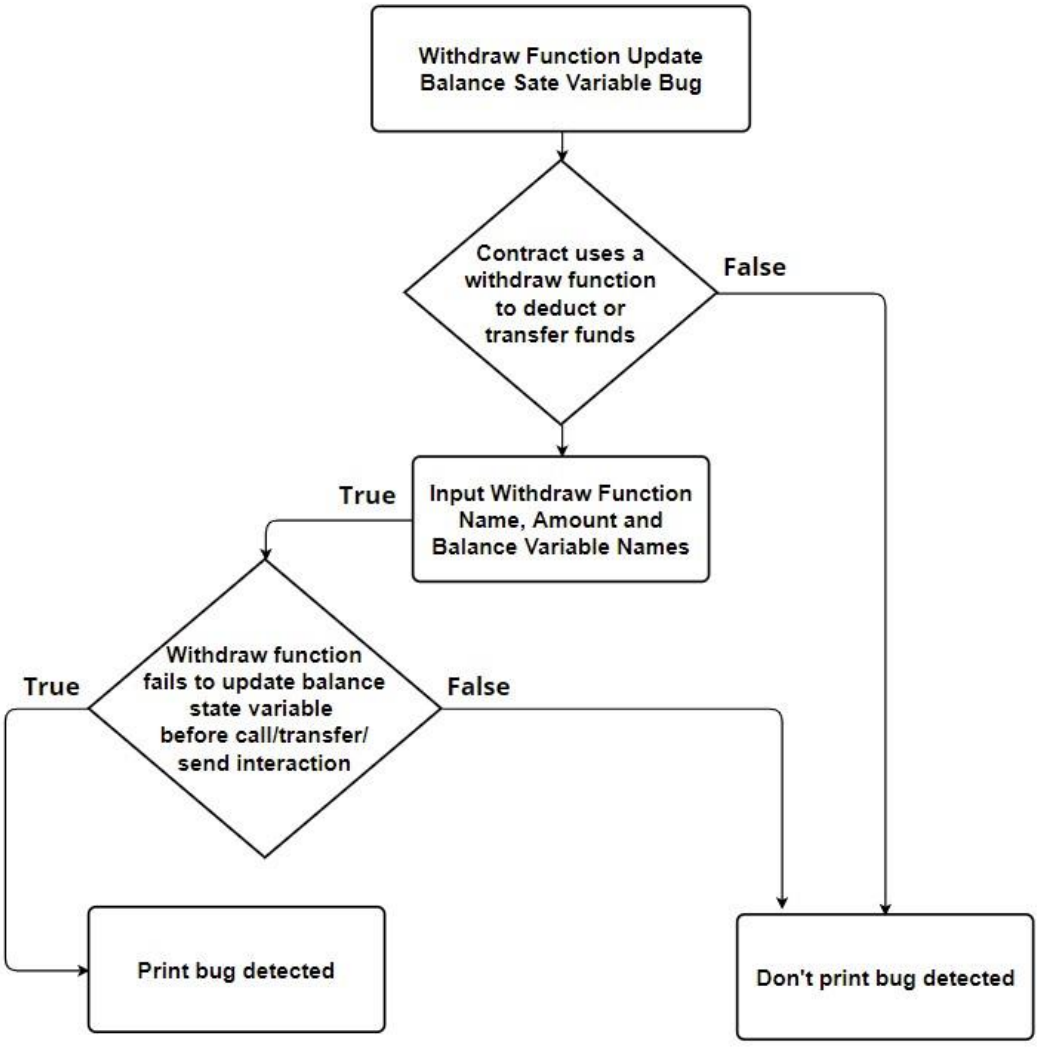
7.1.29 Block Reentracy CFD



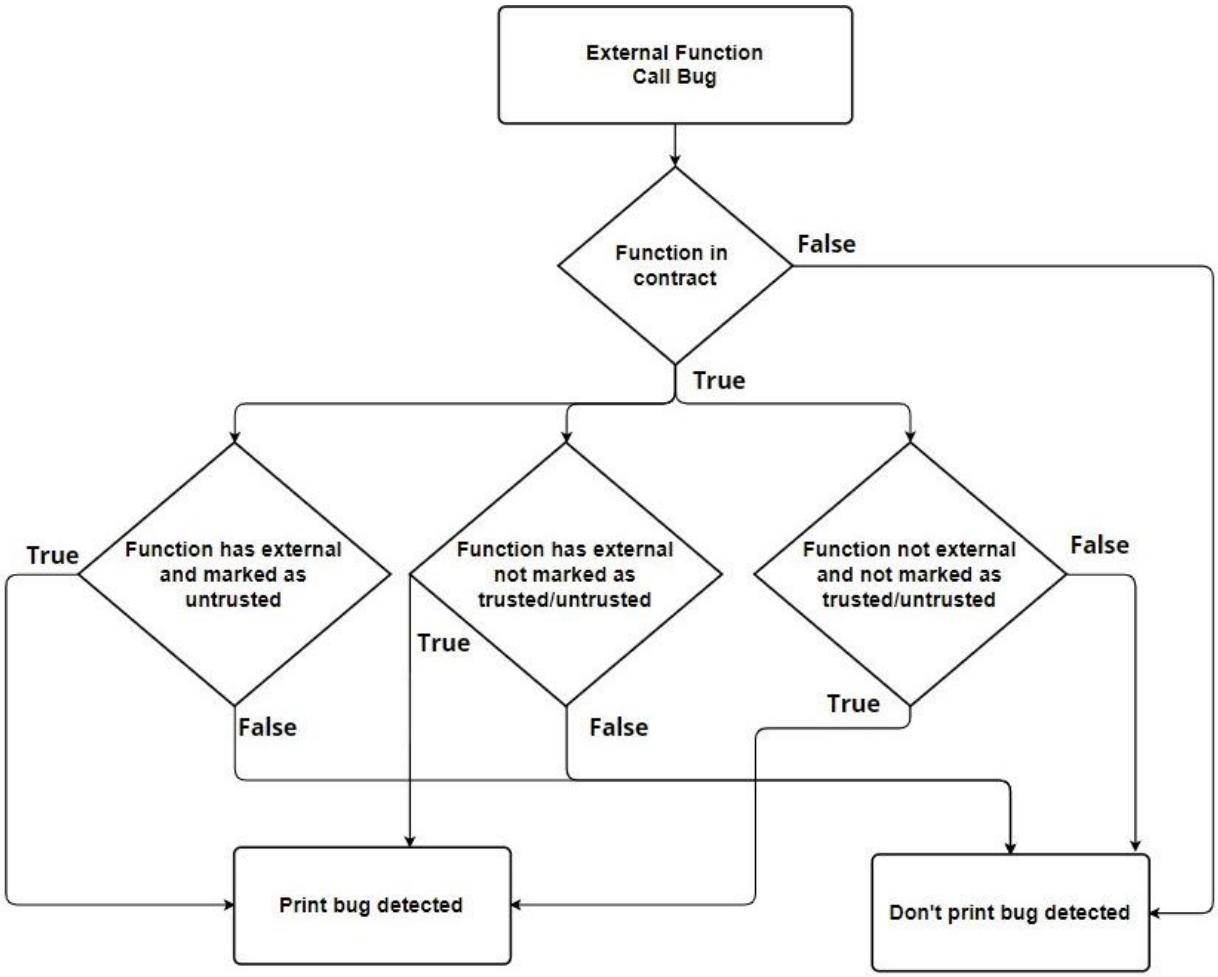
7.1.30 Withdraw Require Condition CFD



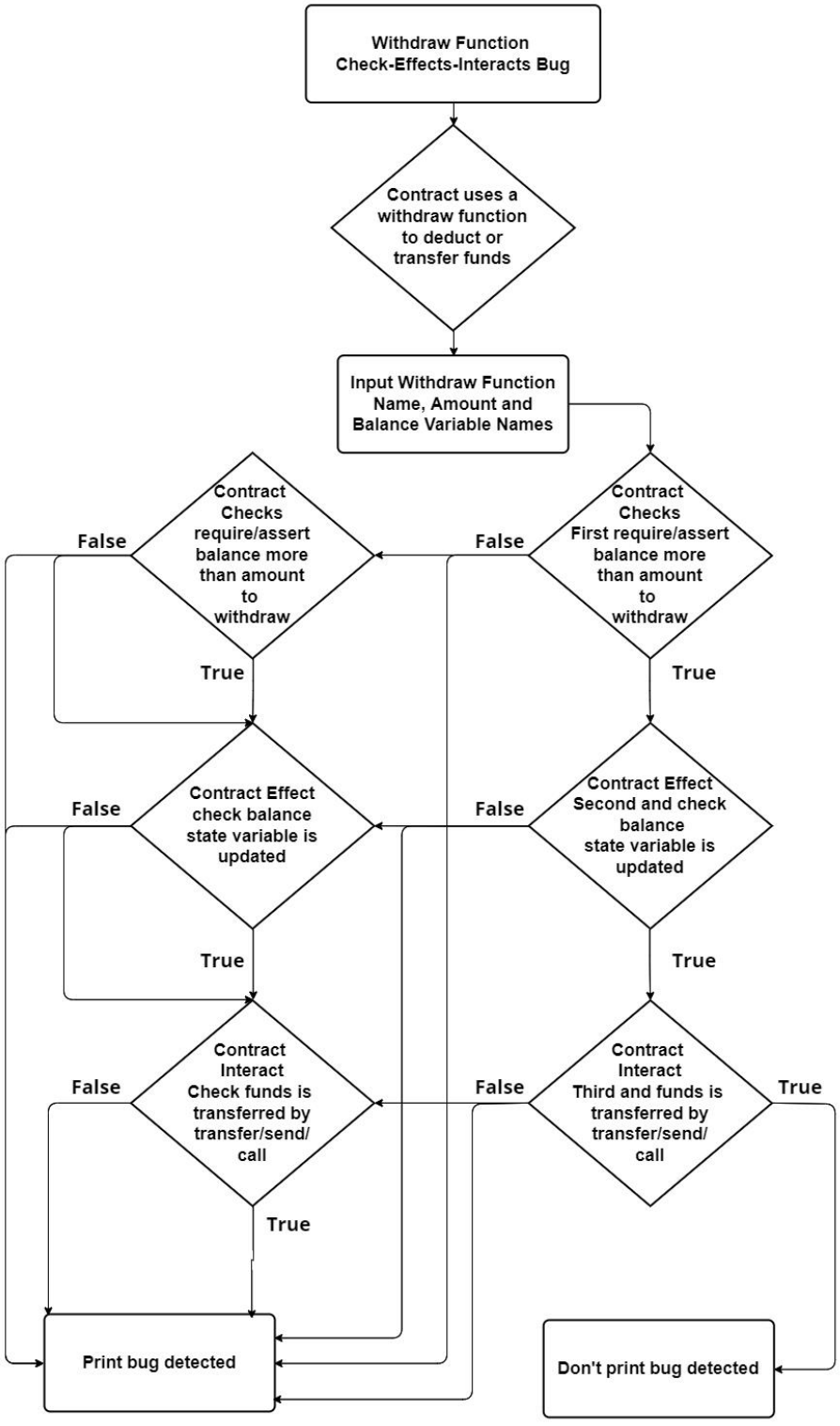
7.1.31 Withdraw Balance State Variable CFD



7.1.32 External Call CFD



7.1.33 Check-Effect-Interact Pattern CFD



7.2 Appendix B – Codebase

All code used in this Thesis project for both the proposed tool PySolSweep as well as the evaluation testing experiment can be located using the following URL:

https://github.com/nikhilsurfingaus/ThesisProject

The README.md file provides a list of descriptions, overview as well as instructions on how to install and run the Solidity Static Analysis Tool PySolSweep.

7.3 Appendix C – Raw Test Results

7.3.1 Solint Raw Test Results

Static Analysis Tool (Standard):

Measure	Result
Bugs Detected	938
False Positive Bugs Detected	89
Overflow/Underflow Bugs Detected	175
Syntax Bugs Detected	764
DAO Bugs Detected	0
Total Verified Overflow/Underflow Bugs	148
Total Verified Syntax Bugs	690
Total Verified DAO Bugs	0
Total Verified Bugs	849
Detection Accuracy (%)	90.51 %
Overflow/Underflow Detection Accuracy (%)	80.57 %
Syntax Detection Accuracy (%)	90.31 %
DAO Detection Accuracy (%)	0 %

Static Analysis Tool (DAO Withdraw Function):

Measure	Result
DAO Withdraw Bugs Detected	67
False Positive Bugs Detected	17
DAO Withdraw Total Verified Bugs	49
Detection Accuracy (%)	73.13 %

7.3.2 Solidity Scan Raw Test Results

Static Analysis Tool (Standard):

Measure	Result
Bugs Detected	708
False Positive Bugs Detected	52
Overflow/Underflow Bugs Detected	142
Syntax Bugs Detected	566
DAO Bugs Detected	0
Total Verified Overflow/Underflow Bugs	129
Total Verified Syntax Bugs	527
Total Verified DAO Bugs	0
Total Verified Bugs	656
Detection Accuracy (%)	92.66 %
Overflow/Underflow Detection Accuracy (%)	90.85 %
Syntax Detection Accuracy (%)	93.11 %
DAO Detection Accuracy (%)	0 %

Static Analysis Tool (DAO Withdraw Function):

Measure	Result
Bugs Detected	7
False Positive Bugs Detected	2
Total Verified Bugs	5
Detection Accuracy (%)	71.43 %

7.3.3 SmartCheck Raw Test Results

Static Analysis Tool (Standard):

Measure	Result
Bugs Detected	1661
False Positive Bugs Detected	249
Overflow/Underflow Bugs Detected	370
Syntax Bugs Detected	1292
DAO Bugs Detected	0
Total Verified Overflow/Underflow Bugs	316
Total Verified Syntax Bugs	1076
Total Verified DAO Bugs	0
Total Verified Bugs	1390
Detection Accuracy (%)	83.68 %
Overflow/Underflow Detection Accuracy (%)	85.41 %
Syntax Detection Accuracy (%)	83.28 %
DAO Detection Accuracy (%)	0 %

Static Analysis Tool (DAO Withdraw Function):

Measure	Result
Bugs Detected	18
False Positive Bugs Detected	4
Total Verified Bugs	12
Detection Accuracy (%)	66.67 %

7.3.4 sFuzz Raw Test Results

Static Analysis Tool (Standard):

Measure	Result
Bugs Detected	3966
False Positive Bugs Detected	468
Overflow/Underflow Bugs Detected	1488
Syntax Bugs Detected	2454
DAO Bugs Detected	41
Total Verified Overflow/Underflow Bugs	1282
Total Verified Syntax Bugs	2160
Total Verified DAO Bugs	26
Total Verified Bugs	3466
Detection Accuracy (%)	87.39 %
Overflow/Underflow Detection Accuracy (%)	86.16 %
Syntax Detection Accuracy (%)	88.02 %
DAO Detection Accuracy (%)	63.41 %

Static Analysis Tool (DAO Withdraw Function):

Measure	Result
Bugs Detected	49
False Positive Bugs Detected	13
Total Verified Bugs	36
Detection Accuracy (%)	73.47 %

7.3.5 Remix Raw Test Results

Static Analysis Tool (Standard):

Measure	Result
Bugs Detected	1748
False Positive Bugs Detected	167
Overflow/Underflow Bugs Detected	554
Syntax Bugs Detected	1194
DAO Bugs Detected	0
Total Verified Overflow/Underflow Bugs	489
Total Verified Syntax Bugs	1090
Total Verified DAO Bugs	0
Total Verified Bugs	1571
Detection Accuracy (%)	89.87 %
Overflow/Underflow Detection Accuracy (%)	88.27 %
Syntax Detection Accuracy (%)	91.29 %
DAO Detection Accuracy (%)	0 %

Static Analysis Tool (DAO Withdraw Function):

Measure	Result
DAO Withdraw Bugs Detected	82
False Positive Bugs Detected	21
DAO Withdraw Total Verified Bugs	61
Detection Accuracy (%)	74.39 %

7.3.6 PySolSweep Raw Test Results

Static Analysis Tool (Standard):

Measure	Result
Bugs Detected	4933
False Positive Bugs Detected	813
Overflow/Underflow Bugs Detected	2083
Syntax Bugs Detected	2526
DAO Bugs Detected	324
Total Verified Overflow/Underflow Bugs	1859
Total Verified Syntax Bugs	2024
Total Verified DAO Bugs	237
Total Verified Bugs	4120
Detection Accuracy (%)	83.52 %
Overflow/Underflow Detection Accuracy (%)	89.24 %
Syntax Detection Accuracy (%)	80.13 %
DAO Detection Accuracy (%)	73.15 %

Static Analysis Tool (DAO Withdraw Function):

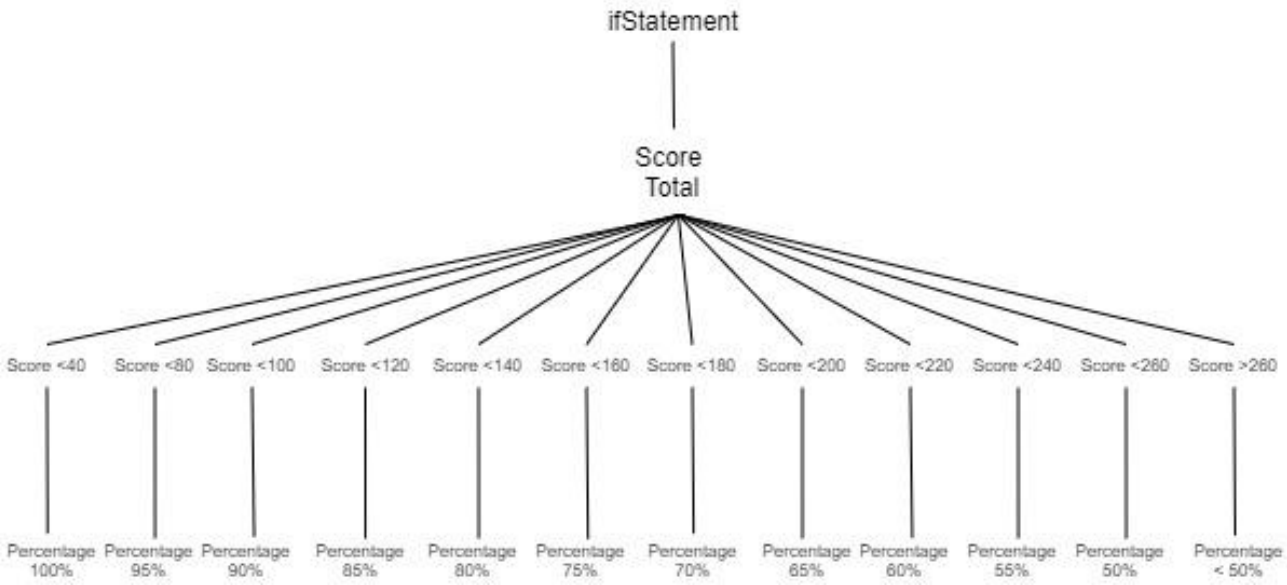
Measure	Result
DAO Withdraw Bugs Detected	230
False Positive Bugs Detected	49
DAO Withdraw Total Verified Bugs	179
Detection Accuracy (%)	77.83 %

7.4 Appendix D – Contract Safety Score Logic

7.4.1 Matrix Score

Risk	x	Confidence
Low (2)		Low (1)
Medium (4)		Medium (2)
High (6)		High (3)

7.4.2 Safety Rating Percentage



8.0 Bibliography

[1]"Program Analysis - Wikipedia", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Program_Analysis. [Accessed: 19- Aug- 2021].

[2]R. Bellairs, "What Is Static Analysis? Static Code Analysis Overview | Perforce Software", *Perforce Software*, 2021. [Online]. Available: https://www.perforce.com/blog/sca/what-Static-Analysis. [Accessed: 19- Aug- 2021].

[3]"Security Tools - Ethereum Smart Contract Best Practices", *Consensys.github.io*, 2021. [Online]. Available: https://consensys.github.io/Smart-Contract-best-practices/security_tools/. [Accessed: 19- Aug- 2021].

[4]"Security testing - Wikipedia", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Security_testing. [Accessed: 19- Aug- 2021].

[5]"What Is Ethereum?", *Investopedia*, 2021. [Online]. Available: https://www.investopedia.com/terms/e/ethereum.asp. [Accessed: 19- Aug- 2021].

[6]G. McCubbin, "The Ultimate Ethereum Dapp Tutorial (How to Build a Full Stack Decentralized Application Step-By-Step) | Dapp University", *Dapp University*, 2021. [Online]. Available: https://www.dappuniversity.com/articles/the-ultimate-ethereum-dapp-tutorial. [Accessed: 19- Aug- 2021].

[7]L. Mearian, "What's a Smart Contract (and how does it work)?", *Computerworld*, 2021. [Online]. Available: https://www.computerworld.com/article/3412140/whats-a-Smart-Contract-and-how-does-it-work.html. [Accessed: 19- Aug- 2021].

[8]"Ethereum price today, ETH live marketcap, chart, and info | CoinMarketCap", *CoinMarketCap*, 2021. [Online]. Available: https://coinmarketcap.com/currencies/ethereum/. [Accessed: 19- Aug- 2021].

[9]"Smart Contract Security: What Are the Weak Spots of Ethereum, EOS, and NEO Networks?", *TechNative*, 2021. [Online]. Available: https://technative.io/Smart-Contract-security-what-are-the-weak-spots-of-ethereum-eos-and-neo-networks/. [Accessed: 19- Aug- 2021].

[10]"The Landscape of Solidity Smart Contract Security Tools in 2020", *Kleros*, 2021. [Online]. Available: https://blog.kleros.io/the-landscape-of-Solidity-Smart-Contract-security-tools-in-2020/. [Accessed: 19- Aug- 2021].

[11]E. Attacks.md, "Ethereum Attacks.md", *Gist*, 2021. [Online]. Available: https://gist.github.com/ethanbennett/7396bf3f61dd985d3426f2ee184d8822. [Accessed: 19- Aug- 2021].

[12]A. John and T. Westbrook, "Crypto platform Poly Network rewards hacker with $500,000 'bug bounty'", *Reuters*, 2021. [Online]. Available: https://www.reuters.com/technology/crypto-platform-poly-network-rewards-hacker-with-500000-bug-bounty-2021-08-13/. [Accessed: 19- Aug- 2021].

[13]"Static Analysis of Integer Overflow of Smart Contracts in Ethereum | Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy", *Dl.acm.org*, 2021. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3377644.3377650. [Accessed: 19- Aug- 2021].

[14]"SmartCheck: Static Analysis of Ethereum Smart Contracts", *Ieeexplore.ieee.org*, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8445052. [Accessed: 19- Aug- 2021].

[15]P. Praitheeshan, L. Pan, J. Yu, J. Liu and R. Doss, "Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1908.08605. [Accessed: 19- Aug- 2021].

[16]"ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks | Proceedings of the 15th ACM Asia Conference on Computer and Communications Security", *Dl.acm.org*, 2021. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3320269.3384756. [Accessed: 19- Aug- 2021].

[17]"How effective are Smart Contract Analysis tools? evaluating Smart Contract Static Analysis tools using bug injection | Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis", *Dl.acm.org*, 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397385. [Accessed: 19- Aug- 2021].

[18]"Smart Contract: Attacks and Protections", *Ieeexplore.ieee.org*, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/8976179. [Accessed: 19- Aug- 2021].

[19]H. Preston, "Integer overflow and underflow vulnerabilities - Infosec Resources", *Infosec Resources*, 2020. [Online]. Available: https://resources.infosecinstitute.com/topic/integer-overflow-and-underflow-vulnerabilities/. [Accessed: 01- Jun- 2022].

[20]D. Siegel, "Ethereum Understanding The DAO Attack", *CoinDesk*, 2022. [Online]. Available: https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/. [Accessed: 01- Jun- 2022].

[21]J. Wu, "Ethereum's History: From Zero to 2.0", *WisdomTree*, 2021. [Online]. Available: https://www.wisdomtree.com/blog/2021-07-15/ethereums-history-from-zero-to-20. [Accessed: 01- Jun- 2022].

[22]M. Staderini, A. Pataricza and A. Bondavalli, "Security Evaluation and Improvement of Solidity Smart Contracts", *SSRN*, 2022. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4038087. [Accessed: 01- Jun- 2022].

[23]"Solidity Scan", *Docs.Solidityscan.com*, 2022. [Online]. Available: https://docs.Solidityscan.com/. [Accessed: 01- Jun- 2022].

[24]"GitHub - Smartdec/Smartcheck: SmartCheck – a Static Analysis tool that detects vulnerabilities and bugs in Solidity programs (Ethereum-based Smart Contracts).", *GitHub*, 2019. [Online]. Available: https://github.com/Smartdec/Smartcheck. [Accessed: 01- Jun- 2022].

[25]"ContractGuard - Testing Platform for Smart Contracts", *Contract.guardstrike.com*, 2019. [Online]. Available: https://Contract.guardstrike.com/. [Accessed: 11- Jun- 2022].

[26]"Security Considerations — Solidity 0.6.11 documentation", *Docs.Soliditylang.org*, 2020. [Online]. Available: https://docs.Soliditylang.org/en/v0.6.11/security-considerations.html. [Accessed: 05- Jun- 2022].

[27]"GitHub - SilentCicero/solint: A linting utility for Ethereum Solidity Smart-Contracts", *GitHub*, 2019. [Online]. Available: https://github.com/SilentCicero/solint. [Accessed: 06- Jun- 2022].

[28]"remix-project/libs/remix-analyzer at master · ethereum/remix-project", *GitHub*, 2020. [Online]. Available: https://github.com/ethereum/remix-project/tree/master/libs/remix-analyzer#readme. [Accessed: 08- Jun- 2022].