Iteration 2 Report

University of Texas at Arlington
Advanced Topics in Software Engineering
Fall 2023, CSE 6324 - 001

https://github.com/PereiraMavs/CSE6324_Team_8

Team 8

Akshata Raikar (1002032638)
Devyani Singh (1001959376)
Ravi Rajpurohit (1002079916)
Shovon Niverd Pereira (1002073941)

Introduction

Blockchain technology is a ground-breaking technology that allows users to communicate without needing a trusted intermediary [1]. A "smart contract" is simply a program that runs on the blockchain. It is a collection of code (its functions) and data (its state) that resides at a specific address [2]. Smart contracts automate the transaction of valuable financial assets. Any kind of error in smart contracts may result in huge financial losses. That is why it is important to test smart contracts thoroughly before deploying them to the blockchain. Smart contracts deployed in Ethereum Blockchain are immutable; therefore, it is imperative to look at vulnerabilities before deploying.

Since each Ethereum transaction requires computational resources to execute, they must be paid for to ensure Ethereum is not vulnerable to spam and cannot get stuck in infinite computational loops. Payment for computation is made in the form of a gas fee [3]. To ensure efficient gas usage it is important to know about excessive use of gas, inefficient code patterns, and precise gas usage amount of solidity functions. Our proposed tool intends to address all these concerns.

Repository

https://github.com/PereiraMavs/CSE6324_Team_8

Version: 2.0

Vision

Smart Contracts with inefficient code can waste resources. Also executing a code that surpasses the gas limit can cause exceptions and halt the program midway. It is always good to look for these gas-related vulnerabilities beforehand. Many static and dynamic analysis tools have been developed to detect this kind of situation [4][5].

There are other vulnerabilities that threaten the safety and security of smart contracts. Among many other tools that detect security risks in smart contracts, Slither is one such tool that uses static analysis techniques to detect almost 88 types of threats. It also supplies different APIs to build other analysis tools using its intermediate representation [6].

The vision of this project is to use Slither to build a unified tool that will detect excessive gas usage and find inefficient codes that waste gas. GasGauge [7] and GASOL [8] tools do something similar but if merged with Slither it will add significant value and make a more powerful tool.

Proposed Features

- Estimate the gas cost of a smart contract function before execution to ensure it does not exceed the limit.
- Analyze code to detect inefficient code so that it can be changed to save gas.
- The codes of the tool will be found in the attached GitHub repository.
- The tool will be console-based.

Project Plan

Following are the features that we will be adding to enhance Slither's capabilities -

- **Fuse multiple For-loops[*presented*]:** While thinking about optimizing for Gas, loops are a good space to work in. When multiple For loops are running for the same length, sometimes they can be optimized by achieving the same task in one loop. In these cases, writing multiple loops can be a bad practice and cost more Gas than necessary. Hence a warning for such scenarios can be helpful in saving time and other resources. We are working on this already and have made significant progress to test this.
  Currently, slither does not handle this. Slither takes care of issues of gas optimization for local variable vs state variable changes etc. So this will be a valuable addition.[4]
- **Map instead of Array[Implemented in Iteration 2]:** In the case of memory access, searching elements, and dynamic sizing, maps are more efficient as compared to arrays in terms of gas usage and time spent. This situation can be identified and the developer can be warned for optimization. We are working on this already and have made progress to implement and test this.[10]
- **Redundant SSTORE[Future Iteration]:** Whenever a disk storage operation takes place, disk access costs significant gas value[4]. This happens for all kinds of read/write operations. To optimize this, disk access should be minimized. We plan to implement this in the 3rd iteration.
- **Calculate Cost of Contract[Future Iteration]:** It is important to know if a smart contract function exceeds the gas allowed for execution or gas limit. This will allow the developer to know the need for optimization before execution through this detector. We plan to implement this in the 2nd iteration.

Development Plan:

Iteration 1 (*presented*)	**Iteration 2 (current)**	**Iteration 3**
● After the inception review and feedback, we decided on all the features (mentioned above) to implement for this project. ● Explored the Slither tool and process for adding new detectors. ● Implemented the first proposed detector for fusible For loops in a contract. ● The roadmap for implementation and testing is clear in this iteration. This will help in getting a boost for future iterations.	● Implement and test the first two features proposed. ● Planned pseudocode for 3rd feature - detect redundant storage use. ● Identify the edge cases for the first two features through testing. ● Document everything for reference. ● Plan for addressing edge cases. ● The roadmap for testing is clearer in this iteration. This helps us to plan a closure for this project.	● Convert the pseudocode for the 3rd feature and test it thoroughly. ● Implement the 4th feature for the overall gas cost estimation of a function. ● Address the edge cases. ● Complete testing of all three features and verify usage for the final time. ● Address all the corner cases if still left in a document. ● Document everything about the project's future scope.

Problems faced till now:

- Faced issues with the makefile for the developer installation process in Slither. We resolved this by debugging the errors and using Stackoverflow.
- As expected, due to a lack of familiarity with the Slither tool, we are facing difficulty in adding the detectors and testing them successfully.
- Matching the Python version with Slither was confusing but was resolved using the virtual environment and *pyenv* tool.
- [Iteration 2]Encountered problems during the testing of the newly added detector on WSL. Despite adding the detector and generating a new build, it failed to activate. This was resolved on another computer when using Visual Studio.
- [Iteration 2]Identifying edge cases for the proposed detectors is a challenge. We are working on this by brainstorming, comparing with competitors, reading research papers, and recognizing the use cases in the coding community.

Problems anticipated for future iterations:

- Introducing new vulnerabilities while implementing new features is an important concern. We plan to do good research and take the support of the TA, Professor, and the developer community to identify cases for this.
- The tool's performance after adding the new features also brings a concern. If the performance is compromised in terms of time, it will require additional time to fix it. We plan to keep the last week of iteration 3 for this.
- Identifying edge cases for the proposed detectors is also something that we anticipate. We rely on brainstorming, and suggestions from classmates, TA, and the professor to address as many cases as possible to make robust solutions.
- Finding out and identifying SSTORE patterns to minimize that may pose a challenge as Slither actually receives solidity files, not bytecode files.

Specification and Design

When a particular detector is used in Slither, the output will be given in the form of console-based text.

- **Input**: Smart contracts in solidity
  The input is in the form of a *.sol (solidity)* file along with the necessary properties to test the added detector.
- **Output**: Analysis results from added detectors
  Console output finds out all the nodes representing loops that can be merged. These nodes can be translated to line numbers for the user to interpret with ease.
- **Exceptional cases**:
  - For now, the fusible loop detector only works with identifiers (eg: for i<n; n is an identifier), but not with member access (eg: for i<arr.length; arr.length is member access).
  - For now, we are designing this detector for loop statement of depth 1 and not for nested loops.
  - Array-to-mapping detectors only suggest to use of mapping. But sometimes it may be impossible to avoid using an array according to use cases. For example when we need the number of elements
- **Dependencies**:
  - Python **3.8.18** is required to run Slither and the added detectors.
  - Solidity version **0.8.18** is recommended to compile smart contracts within Slither. It can be changed using *solc-select* which is preinstalled with Slither.
- **How to add detectors:**
  Create a file for the detector(*fusible_loop.py*). A class with AbstractDetector was implemented and *_detect()* abstract method needs to be implemented. After the detector logic is added the detector import should be added in the *all_detectors.py*. After rebuilding the project the new detector will be ready to use.

Use case diagram:

Here is a detailed use case diagram representing the user flow for all the features proposed for this project.

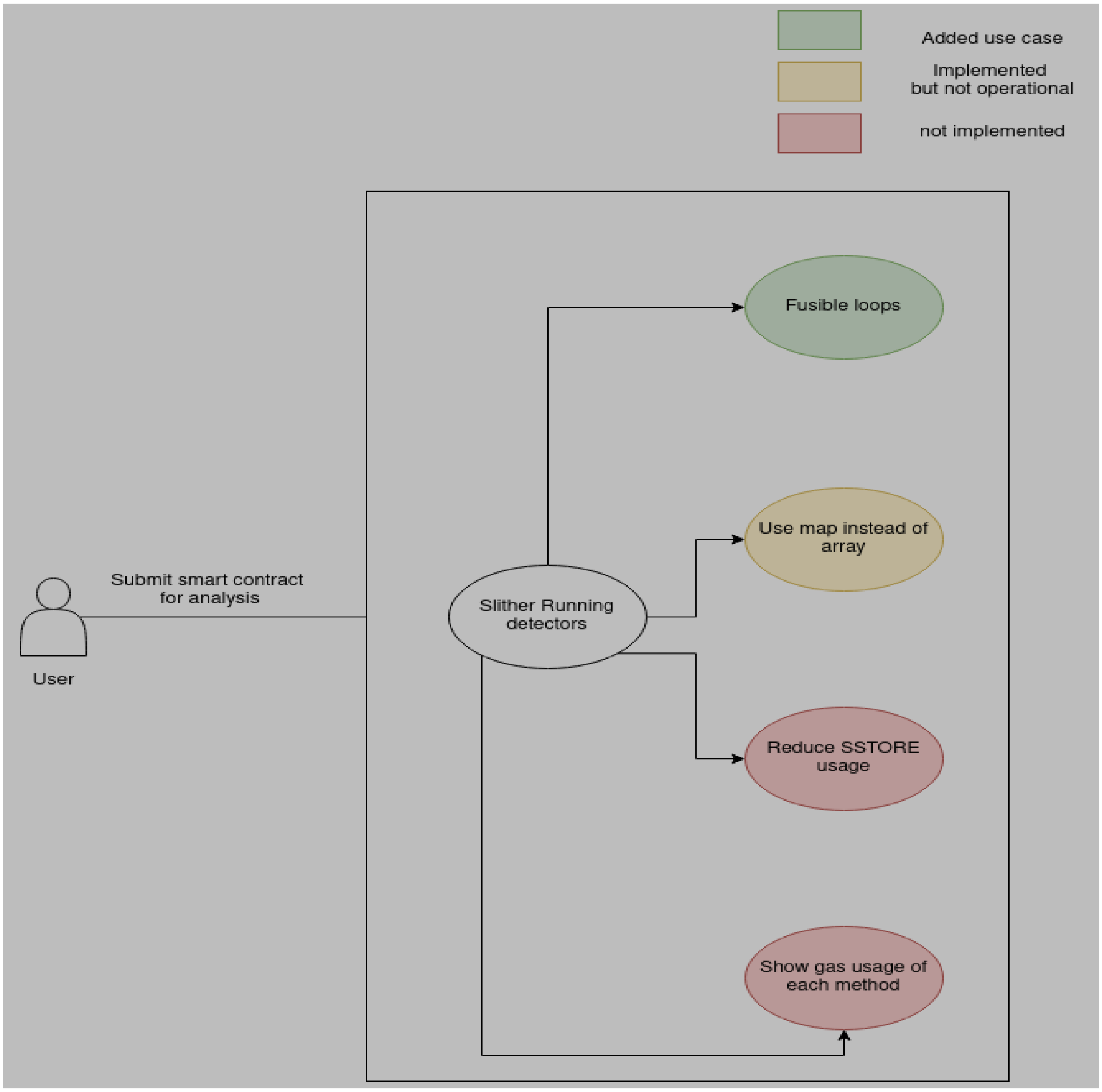

Figure 1: use case diagram

Code and Tests

Detector 1 - Fuse multiple For-loops: This detector has been added to slither, and tested against sample smart contracts. The sample contract is as follows:

loop.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract Loop {
    function loop() public pure {
        // for loops with similar iteration conditions
        uint n = 10;
        for (uint i = 0; i < n; i++) {
            if (i == 3) {
                continue;
            }
            if (i == 5) {
                break;
            }
        }

        for (uint i = 0; i < n; i++) {
            if (i == 3) {
                continue;
            }
        }
    }
}
```

Figure 2: example contract for detector 1 - fuse multiple for loops

The detector relied on finding the nodes in a function that identifies as START_LOOPS. Slither provides objects to identify these nodes. After that terminating conditions were extracted and checked if two loops use the same terminating condition. For now, this detector can only detect fusible loops if they use the same identifier as the right operand of the terminating condition. Code snippets and results are added.

```python
#finds the loops that can be merged
#returns a list of tuples of the loops that can be merged
#the tuple contains the nodes of the loops
#the first node of the tuple is the first loop
#the second node of the tuple is the second loop
@staticmethod
def _findLoop(nodes: List[Node], fuse: List[Node]):
    visited = []
    for node in nodes:
        if node.type == NodeType.STARTLOOP:
            if_node = node.sons[0]

            exp: BinaryOperation = if_node.expression
            visited.append(node)
    for i in range(0,len(visited)):
        for i2 in range(i + 1,len(visited)):
            n1: Node = visited[i]
            n2: Node = visited[i2]
            e1: BinaryOperation = n1.sons[0].expression
            e2: BinaryOperation = n2.sons[0].expression
            if str(e1.expression_right) == str(e2.expression_right):
                fuse.append((n1, n2))

    return fuse
```

Figure 3: detector 1 code

```python
#entry point of the detector
#calls the _findLoop function to find the loops that can be merged
def _detect(self) -> List[Output]:
    results = []
    funtions = self.compilation_unit.functions

    issue = []
    for f in funtions:

        issue += FusibleLoops._findLoop(f.nodes, [])

    for i in issue:
        info = [
            "Loop condition ",
            i[0],
            i[1],
            " can be merged into one loop.\n",
        ]
        res = self.generate_result(info)
        results.append(res)
    return results
```

Figure 4: entry point for detector 1 in slither

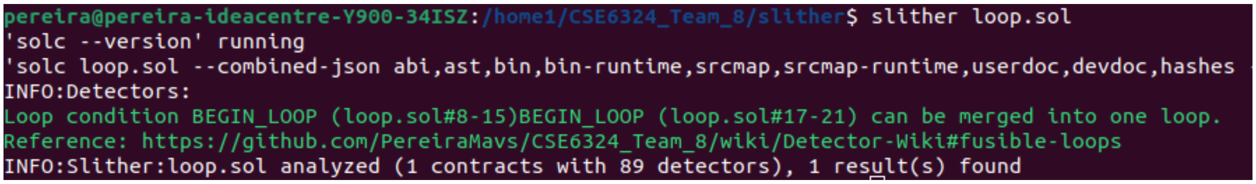

Figure 5: console output of detector 1 - fusible for loops

Goal of detector 1 - Gas Consumption of multiple loops

We can play around with the gas consumption for a smart contract at *evm.codes/playground*

The following images represent the benefits of merging unnecessary loops.[9]

Figure 6: Gas consumption with MULTIPLE for loops

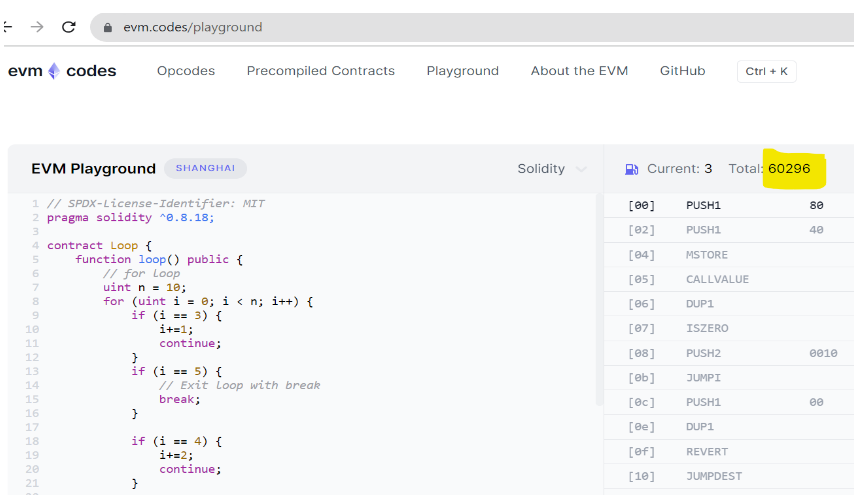

Figure 7: Gas consumption with FUSED for loops

Detector 2 - Map instead of Array: There are various scenarios identified where using a Map data structure is a significantly better decision than using an array. This detector has been added to slither tested against sample smart contracts. The sample contract is as follows:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract ArrayUsageExample {
    uint[] public dynamicArray;
    uint[5] public fixedArray;
    mapping(address => uint) public map;

    constructor() {
        dynamicArray.push(1);
        dynamicArray.push(2);
        dynamicArray.push(3);

        fixedArray[0] = 10;
        fixedArray[1] = 20;
        fixedArray[2] = 30;

        map[msg.sender] = 100;
    }

    function addValueToDynamicArray(uint _value) public {
        dynamicArray.push(_value);
    }

    function getValueFromDynamicArray(uint _index) public view returns (uint) {
        require(_index < dynamicArray.length, "Index out of bounds");
        return dynamicArray[_index];
    }
}
```

Figure 8: entry point for detector 2

The detector examines each contract's state variables, identifies arrays, and suggests using mapping instead for gas efficiency and better performance.

```python
@staticmethod
def _findArraysAndSuggestMaps(contract: Contract):
    issues = []

    #Iterate through the state variables of the contract:
    for state_variable in contract.variables:
        #Check if the state variable's type is an array (ArrayType):
        if (isinstance(state_variable.type, ArrayType)):
            #If an Array state variable is found, add it to the list of issues:
            issues.append(state_variable)

    return issues
```

Figure 9: detector 2 code

```python
def _detect(self) -> List[Output]:
    results = []


    for contract in self.contracts:
        #Find array state variables in the contract:
        array_variables = self._findArraysAndSuggestMaps(contract)

        for array_variable in array_variables:
            #Generate a result for the detected issue:
            info = [f"In contract `{contract.name}`, for variable `{array_variable.name}`
                    ({array_variable.type}), consider using a mapping instead of an array."]
            res = self.generate_result(info)
            results.append(res)

    return results
```

Figure 10: entry point for detector 2

Following is the snippet of the output:

```
pereira@PereirasLaptop:~/CSE6324_Team_8/slither$ slither arraymap.sol --detect array-to-mapping
'solc --version' running
'solc arraymap.sol --combined-json abi,ast,bin,bin-runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes --allo
w-paths .,/home/pereira/CSE6324_Team_8/slither' running
INFO:Detectors:
In contract `ArrayUsageExample`, for variable `dynamicArray` (uint256[]), consider using a mapping instead of
 an array.In contract `ArrayUsageExample`, for variable `fixedArray` (uint256[5]), consider using a mapping i
nstead of an array.Reference: https://github.com/PereiraMavs/CSE6324_Team_8/wiki/Detector-Wiki#fusible-loops
INFO:Slither:arraymap.sol analyzed (1 contracts with 1 detectors), 2 result(s) found
```

Figure 11: Console output of detector 2

Goal of detector 2 - Gas Consumption: Map Vs Array

We can play around with the gas consumption for a smart contract at evm.codes/playground

The following snippets show the difference in gas consumption upon using array and mapping.

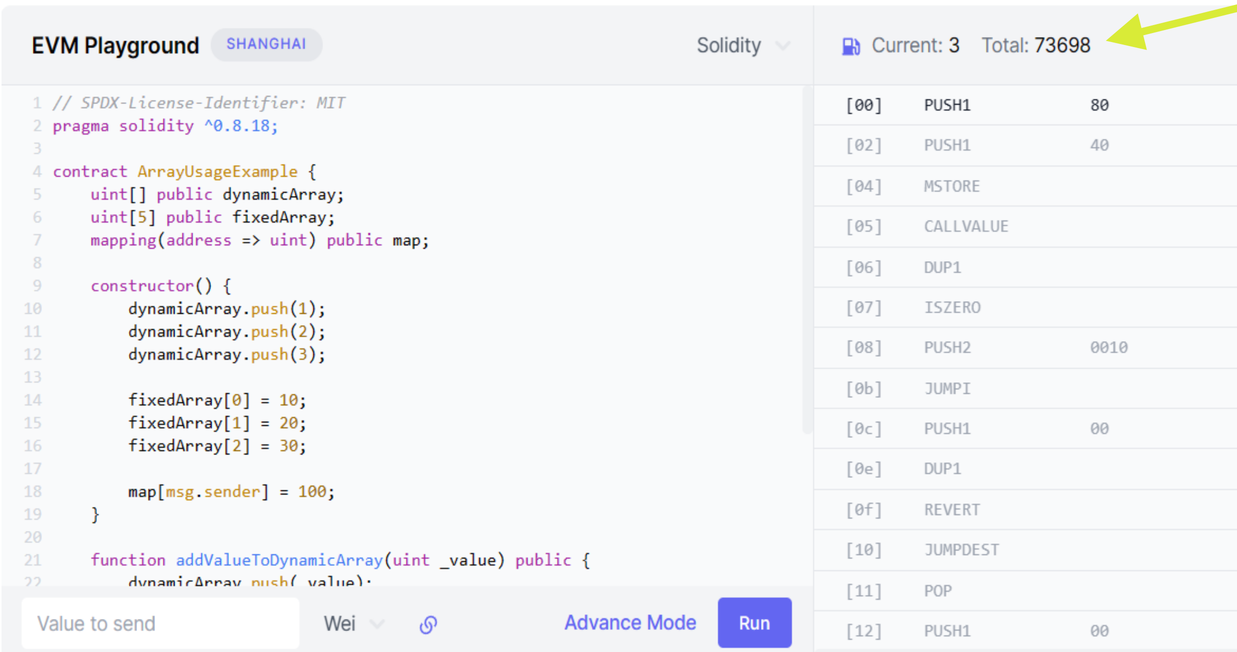

Figure 12: Gas consumption with arrays

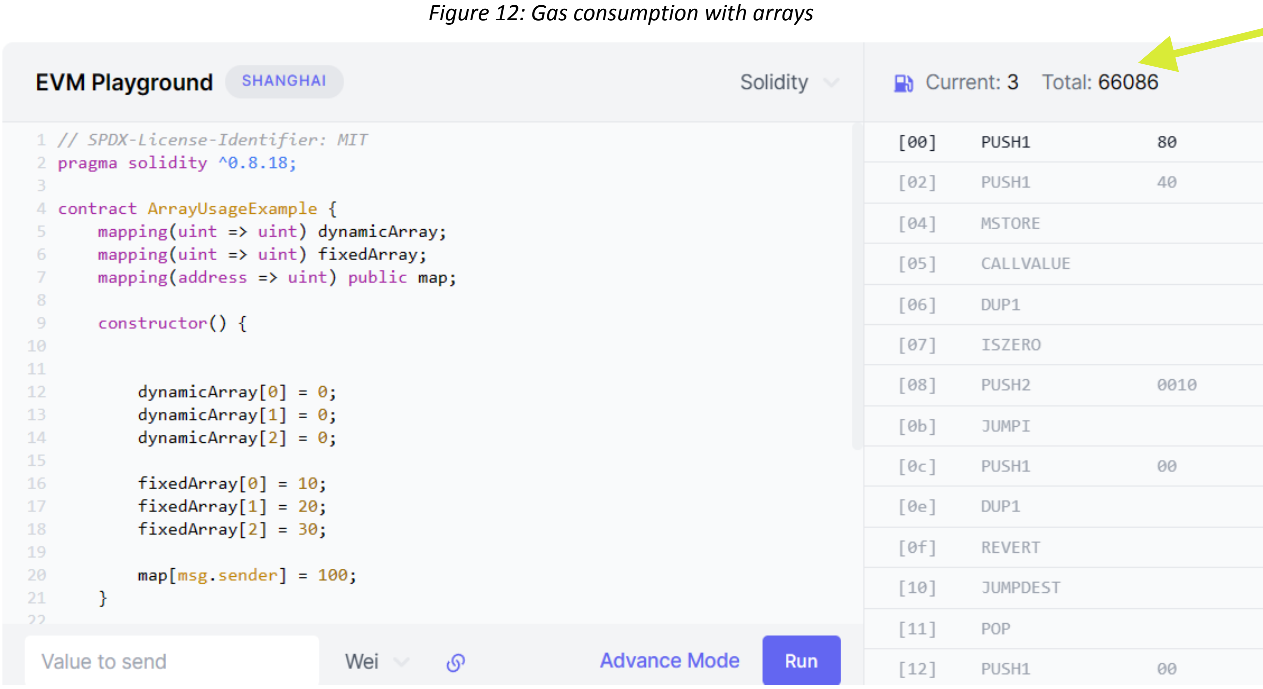

Figure 13: Gas consumption with mapping

Detector 3 - Redundant SSTORE Operation:

The goal of this detector is to find out redundant storage operations. When a storage is never used after definition we can consider that a redundant storage operation. The definition of a storage indicates the opcode SSTORE[4]. SSTORE opcode uses a minimum of 100 units of gas[9]. That means if a word is saved in a storage but never used in the execution path of the program then the former will be redundant.

Figure 11 shows that in lines 110 and 116 a storage operation was done but the left operand variable was never used.

```
55   uint baseprice;
     ...
108  function setBasePrice(uint new_baseprice){
109     if((msg.sender != owner) && (msg.sender != cbAddress)) throw;
110     baseprice = new_baseprice;
111     for(uint i=0; i < dsources.length; i++) price[dsources[i]] = new_baseprice *
                price_multiplier[dsources[i]];
112  }
113
114  function setBasePrice(uint new_baseprice, bytes proofID){
115     if((msg.sender != owner) && (msg.sender != cbAddress)) throw;
116     baseprice = new_baseprice;
117     for (uint i=0; i < dsources.length; i++) price[dsources[i]] = new_baseprice *
                price_multiplier[dsources[i]];
118  }
```

Figure 14: Exploit scenario of redundant SSTORE operation[4]

A brief pseudocode looks like the following -

```
function _findRedundantSSTORE()
    list1 = findAssigmentOperations()
    for all operations in list1
        if operation in storage operation
            result += findIfStorageNeverUsed
    return result
```

Figure 15: Pseudocode for detector 3

To detect if a storage operation is redundant all program execution paths need to be checked. At the moment slither APIs allow us to detect any assignment operation that will primarily help us find storage operations. But we are working on how to ensure we have checked all the execution paths so that it can be said surely if a storage operation is redundant or not.

Customers and Users

Slither is a smart contract security framework written in Python and is widely used in the blockchain developer community. While identifying potential customers and users for this project, we came across the following points -

- Blockchain Developers - developers working on Ethereum, Binance smart chain, or other blockchain platforms who create smart contracts. They use security analysis tools to identify vulnerabilities and improve their code.
- Auditors for Smart Contracts - Professionals or companies specializing in smart contract security audits. They require advanced tools to assess the security and reliability of smart contracts.
- Academic Researchers - Academics and researchers in the field of blockchain and smart contract security who could use the tool for their studies and experiments.
- Blockchain Enthusiasts - Individuals interested in blockchain technology who develop smart contracts as a hobby or for personal projects. They may use the tool for learning and experimentation.

Competitors

Tool	**Features**
GasGauge [7]	GasGauge focuses on gas-related vulnerabilities that occur by out-of-gas situations
GasFuzzer[5]	It improves on the ContractFuzzer tool to find out vulnerabilities via gas allowance manipulation
GasChecker[4]	Suggests efficient smart contract coding technique to improve gas usage

Risk Management Plan

- **Technical Risk:** As we plan to try uniting the functions of different tools, there is a risk that the process might throw some unanticipated errors that will need some time to be corrected. While progressing through the iterations, we are identifying a variety of technical risks.
- **Resource Risk:** Most of the team lacks familiarity and is inexperienced with Smart Contracts to work smoothly on such a project. It might take some extra time than a standard plan to fill this gap.
  We are also facing issues while trying to add the detector with Slither because of version dependencies.
- **Schedule Risk:** All the team members have different class schedules and varied commitments. Finding a disciplined yet flexible schedule is a challenge due to conflicting classes and assignments.

Risk Mitigation Plan

- **Technical Risk Mitigation:** We are taking help from the wide and supportive Python community along with capable AI resources like chatGPT to explore the scope of problems and ideas. Moreover, by following the best coding practices, we will reduce the odds of errors and increase the chances of success as defined in the project description.
- **Resources Risk Mitigation:** As we already know about lack of familiarity with such projects, all the team members are expanding their knowledge through lectures and research papers relating to the relevant topics for this subject. We are giving extra time to catch up with the inexperience in this field.
- **Schedule Risk Mitigation:** As a team, we are consistently planning to be flexible with the project development schedule for each member. The plan includes keeping coherent and regular communication throughout the semester. Keeping a slack of time for each stage is proving to be beneficial in dealing with unexpected issues.

Risk Exposure

Considering each risk category, we believe that Technical and Resource-based risks will take the most time to resolve. While Schedule risks are significant too, its weightage is proven, till now, to be lesser than other categories.

- Technical Risk Exposure
    - Occurrence probability (PR) - 0.5
    - Occurrence effect (ER) - Extra 15 hours
    - Exposure of risk – 0.5 (PR) * 15 (ER) = 7.5 hours
- Resource Risk Exposure
    - Occurrence probability (PR) - 0.5
    - Occurrence effect (ER) - Extra 12 hours
    - Exposure of risk – 0.5 (PR) * 12 (ER) = 6 hours
- Schedule Risk Exposure
    - Occurrence probability (PR) - 0.2
    - Occurrence effect (ER) - Extra 8 hours
    - Exposure of risk – 0.2 (PR) * 8 (ER) = 1.6 hours

Hence, with the iteration, we considered these risk exposures and effectively tried to plan better for the project's final progress till now.

References

[1] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur and H. -N. Lee, "Ethereum Smart Contract Analysis Tools: A Systematic Review," in IEEE Access, vol. 10, pp. 57037-57062, 2022, doi: 10.1109/ACCESS.2022.3169902.

[2] https://ethereum.org/en/developers/docs/smart-contracts/ , accessed on 09/23/2023

[3] https://ethereum.org/en/developers/docs/gas/ , accessed on 09/23/2023

[4] T. Chen et al., "GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts," in IEEE Transactions on Emerging Topics in Computing, vol. 9, no. 3, pp. 1433-1448, 1 July-Sept. 2021, doi: 10.1109/TETC.2020.2979019.

[5] I. Ashraf, X. Ma, B. Jiang and W. K. Chan, "GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities," in IEEE Access, vol. 8, pp. 99552-99564, 2020, doi: 10.1109/ACCESS.2020.2995183.

[6] https://github.com/crytic/slither, accessed on 09/23/2023

[7] https://arxiv.org/abs/2112.14771, accessed on 09/23/2023

[8] Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A. (2020). GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts. In: Biere, A., Parker, D. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2020. Lecture Notes in Computer Science (), vol 12079. Springer, Cham. https://doi.org/10.1007/978-3-030-45237-7_7

[9] https://www.evm.codes/

[10] https://www.linkedin.com/pulse/how-reduce-smart-contract-gas-usage-arslan-maqbool