GasCore+ Iteration 2

Team 5

Team Members: Abbinav H. Burhanuddin C. Dhairya C. Ritesh M.

CSE 6324-001

**Project Plan**

*Features Plan*

* Change Gas Gauge and Manticore libraries to run on WSL.
  + The first goal of the team’s feature plan is to integrate Manticore and Gas Gauge into WSL. The team will use WSL to run both Manticore and Gas Gauge tools. Both Manticore and Gas Gauge use similar supporting libraries, but both tools use different versions of those supporting libraries. In order for both tools to work on WSL, the team will update the supporting libraries to a version that is compatible with both Manticore and Gas Gauge.
* Integrate Manticore API with Gas Gauge ..
  + The second goal of the team’s feature plan is to incorporate the Manticore API into Gas Gauge. The team will use the Manticore API to add features to Gas Gauge instead of the full tool. The Manticore API features allows a better compatibility with Gas Gauge. Instead of the team importing thousands of lines of code for Manticore features, the API will allow us to mimic the functionality of Manticore. The API itself is very extensive and allows to accomplish a lot of functionalities of Manticore rather than integrating a lot of code directly into Gas Gauge to accomplish those same features.
* Incorporate Manticore Features (Python Code) with Gas Gauge Features (Python Code) for GasCore+ Base.
  + The third goal of the team’s feature plan is to combine the two tool’s Manticore and Gas Gauge features and start the GasCore+ base. After understanding the Manticore features and making those features compatible with Windows OS, the team will be able to slowly implement the features of both tools together while making sure the tool still functions properly. The results will be compared to the original two tool’s results to ensure validity of those results. Shovon will assist the team in the process of comparing and verifying the results between GasCore+ and Gas Gauge/Manticore.

*App Differences*

* Slither: Incorporates Static Analysis
  + Slither uses static analysis to help enhance vulnerability detection, optimization detection, code understanding and assisted code review. Slither takes in initial input and generates control flow graphs, contract inheritance graphs and a list of expressions within the contract. Next Slither translates the code into SlithIR, an internal language to make it easier to analyze and write. Finally Slither runs predefined analyses that provide enhanced information such as computing data flow and protected function calls, etc. Slither however does not execute dynamic analysis nor symbolic execution like the GasCore+ is aimed to achieve. [1]
* GasCore+: Incorporate Static Analysis, Dynamic Analysis and Symbolic Execution from Gas Gauge and Manticore.
  + GasCore+ uses both static and dynamic analysis and then runs symbolic execution to help enhance the capabilities of finding bugs in Ethereum Smart Contracts. Unlike Slither, GasCore+ also uses the dynamic analysis efficiency of Gas Gauge to help analyze smart contracts and then runs it through MantiCore’s symbolic execution process to help eliminate as many vulnerabilities in the Ethereum Smart Contract as possible. The extra dynamic analysis of Gas Gauge provides an extra level of analysis that Slither does not provide, making GasCore+ an all-rounder in Fuzz Testing and Symbolic Execution.

*5 Biggest Risks*

* Risk 1: Incorporating Manticore Features with Gas Gauge Features.
  + Adding Manticore features to Gas Gauge features could cause compile and runtime errors when trying to form the base of GasCore+ as both features are completely different from each other.
  + **Risk Exposure:** The likelihood of this risk occurring is 60% and the impact on this iteration will be about 10 hours spent. The Risk Exposure for Risk 1 is about 6 extra hours spent towards resolving it.
  + **Mitigation Plan:** The team has discussed with Shovon to try to implement simple features of Manticore first due to the complexity of features in the tool. Once simple features are implemented, other features can be considered with respect to the schedule delivery at the end of the semester.
* Risk 2: Integrating Manticore and Gas Gauge libraries to be compatible with WSL.
  + Manticore currently works with Linux OS only while Gas Gauge works with both Linux and Windows OS. The team plans to integrate both Gas Gauge and Manticore to be compatible with WSL in order to combine it with the Gas Gauge Tool to form the base of GasCore+.
  + **Risk Exposure:** The likelihood of this risk occurring is 50% and the impact on this iteration will be about 6 hours spent. The Risk Exposure for Risk 2 is about 3 extra hours spent towards resolving it.
  + **Mitigation Plan:** The team will try to mimic the functionality of Manticore features into Gas Gauge working on Windows OS through WSL. The team will revert back to Linux OS in case Manticore proves to be more difficult to convert to WSL than anticipated.
* Risk 3: Understanding Manticore API
  + Manticore is a very complex tool used to analyze smart contracts. The team will try to understand the Manticore API and only incorporate features that are available through the API.
  + **Risk Exposure:** The likelihood of this risk occurring is 40% and the impact on this iteration will be about 7.5 hours spent. The Risk Exposure for Risk 3 is about 3 extra hours spent towards resolving it.
  + **Mitigation Plan:** The team will raise tickets on the Manticore Github page with questions about the Manticore API and how the features of the API work.
* Risk 4: Running Smart Contracts through GasCore+.
  + Finding and running smart contracts through GasCore+ could cause invalid results when compared to running those same smart contracts through Gas Gauge and Manticore separately.
  + **Risk Exposure:**The likelihood of this risk occurring is 40% and the impact on this iteration will be about 7 hours spent. The Risk Exposure for Risk 4 is about 2.8 extra hours spent towards resolving it.
  + **Mitigation Plan:** The team will work with Shovon to analyze the results of the three tools and discuss where results are valid and where the results could have possibly gone invalid in GasCore+ compared to Gas Gauge and Manticore to better pinpoint the issues in the code/features for GasCore+.
* Risk 5: Team Members’ Scheduling Issues.
  + Team members have homework assignments, projects, and quizzes to work on during the current iteration. Few team members also work during the week reducing the potential time frame of implementing features for GasCore+.
  + **Risk Exposure:** The likelihood of Risk 5 occurring is 30% and the impact on this iteration will be about 6 hours spent. The Risk Exposure for Risk 5 is 1.8 extra hours towards resolving it.
  + **Mitigation Plan:** The team members can work with the team to reschedule meetings and reallocate iteration tasks to team members who can handle a heavier load this iteration than team members who are busy with other class work with similar due dates as the current iteration. Team members can also reduce work hours if applicable.

**Specification and Design**

**Gas Gauge**:Gas Gauge is a hybrid of static and dynamic analysis that focuses on gas-related vulnerabilities and ensures that Solidity gas does not exceed the gas limit during transactions. [4]

**Working of Gas Gauge**: Gas Gauge operates in three stages: Detection, Identification, and Correction. The Detection Phase employs a static analysis approach to find any loops in a smart contract in a quick and accurate manner. During the Identification Phase, a white-box fuzzing approach is used to generate a set of inputs for each user-controlled loop in a Solidity smart contract's public function. The Identification Phase allows you to scan contracts before deployment to see if they are at risk of DoS due to Block Gas Limit. The Correction Phase is intended to determine the upper bound limit of a smart contract's loops. The result is a formula based on the maximum number of loop iterations that can be performed before the transaction runs out of gas. [4]

**Output of Gas Gauge**: The Gas Gauge result finds the loops and the number of loops in each function. To produce a set of inputs for each user-controlled loop in a public function within a solidity smart contract, a white-box fuzzing approach is used. The limits of these loops must be impacted by at least one of the function's input variables; otherwise, directly fuzzing the target function is ineffective. When the first modified contract is received, the Threshold Estimator immediately generates a Solidity test file as well as all of the other files required by Truffle Suite. Each loop is run twice in the modified contract, and the gas utilized in each iteration is captured. [4]  
  
Threshold = (gasleft() - gas 1)/(gas 2 + Internal)  
  
The output of the correction phase provides the tool's threshold for the given gas limit, as well as the gas consumption of the first iteration and the average gas consumption of the subsequent iterations. Where gasleft() is the value returned by the function gasleft(), which is placed immediately before the loop in the source code, gas 1 is the gas consumption of the first loop iteration, gas 2 is the average gas consumption of the other iterations, and Internal is the gas consumption of the internal loops in the case of nested loops. Otherwise, it is zero. [4]

**Manticore:** Manticore is a symbolic execution tool for the analysis of smart contracts. It executes a program with symbolic inputs and explores all possible states it can reach.

**Working of Manticore:** Manticore sets up an emulator of a blockchain environment. Once the input contract is given, manticore sends various symbolic transactions to the contract. If a way to break any property in the contract is found then manticore outputs the full transaction tracing. Manticore runs until any of its stopping conditions is reached, if all properties run without failing till the end condition is reached then they are considered to pass. [5]

**Input:** Contract-Name - The name for the contract where the properties that need to be tested are present.

One can also specify what are the accounts used for contract Exploration. There are 3 types of user accounts.

Deployer - The account used to create the Target Contract’

Senders - A set of accounts that send symbolic transactions to the contract.

pSender - Account used as a caller, that actually sends the transactions to test the property methods.

**Output of Manticore:** The output of Manticore is put into a folder. In the folder a *.summary* file summarizes all the vulnerabilities in the smart contract if any are found. There will also be different output reports for each test case. Then manticore also outputs the number of instructions that were executed without breaking their property.

**GAS GAUGE:**

***Input:***The input to the GasCore+ is similar to Gas Gauge in which we will provide the user the opportunity to define the below inputs:  
  
**CONTRACT\_PATH** - The path where the contract to be analyzed is present.

**GAS\_LIMIT** - The Gas Limit that the user can provide.

**FUZZER** - An option for users to enable/disable Fuzzer-based testing.

**THRESHOLD** - An option for users to enable/disable Threshold based testing.

**REPORT\_PATH** - The path where the report of the analysis will be generated.

**FUZZER\_MAX\_ITERATION** - The max iterations of the analysis being done.

**solc = "0.8.1"** - The version of Solidity is based on the user's preference.

***Output:***Prior to running GasCore+, the user has to specify the below configurations:  
  
**output\_folder** - The folder where the output of the analysis will be stored.

**output\_file** - The name of the txt file that will be created.

**gas\_gauge\_dir** - Directory where the contract output will be stored.

**new\_contract\_name** - The name of the contract that will be stored after the analysis.

The above configurations help in analyzing the smart contracts and provide us with an output of the fuzzing testing and show a list of issues that a contract might have.

***Implemented Feature:***

The first use case of this iteration was to ensure that we’re able to run the functionalities of both Gas Gauge and Manticore simultaneously on the same machine. We installed Windows Subsystem for Linux and configured both the applications to be run on the same machine. Rather than copying over the entire codebase of Manticore to Gas Gauge or vice-versa which would increase the complexity of our application and development, we decided to leverage the Manticore API. Utilizing the Manticore API, we were able to develop features of Manticore within our Gas Gauge application. Manticore API allowed us to create a Manticore EVM and utilizing the verifier we were able to pass our contract as an input.  
  
  
***Future Features:***

After integrating and implementing parts of the MantiCore-verifier, we would like to test the application with several scenarios within the verifier such as

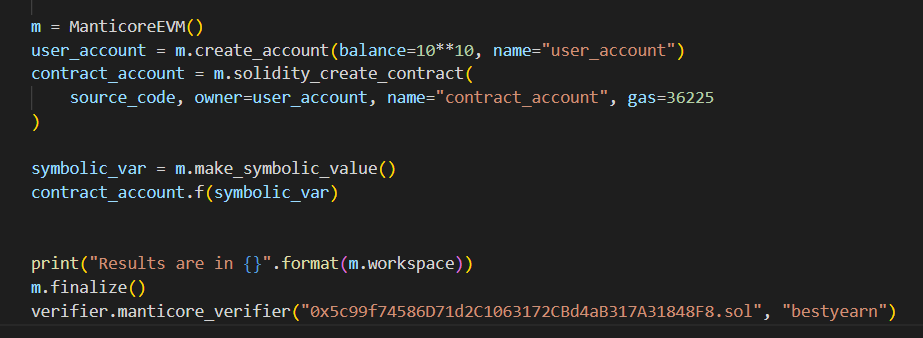
* User Contracts - Allows us to define the account of the developer, sender, and psender of the contract which aids in security and prohibits several security vulnerabilities. [2]
* Stop Conditions - Ensures that the exploration of contracts would be stopped after reaching certain exit conditions. [2]

Additionally, we would like to integrate the main Manticore functionality with Gas Gauge. We were able to build the basic structure for Manticore and are able to test one function within a Contract. We would like to extend this functionality to multiple functions and then eventually an entire contract and provide symbolic execution.

**Code and Tests**The code changes done as part of this iteration are available through our public repository - <https://github.com/Abbinav/CSE-6324-Team5/tree/master>

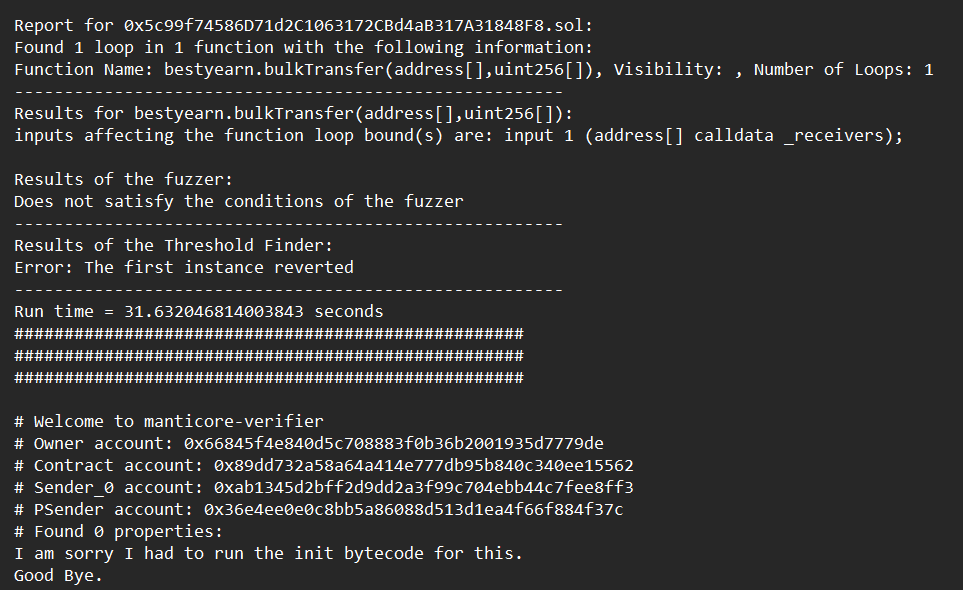
**Code/Configuration Changes Done:**

* Modifying Slither Version to ensure both Gas Gauge and Manticore can co-exist
* Upgrading Crytic-Compile version to run both Apps
* Integrating Manticore within GasGauge
  + Initiating a MantiCoreEvm by importing Manticore API library within Gas Gauge. [2]
  + Creating an user account with balance and user account. [3]
  + Utilizing the user account to create a solidity contract. [3]
  + Utilizing symbolic\_value to add constraints and conditions. [3]



The steps required to run and test the code would be as follows:

* Download the code from the repository above.
* Install the libraries provided within the Readme file of the repository.
* Either utilize the sample contract provided within the project or add your own preferred contract to be analyzed:
  + Modify the contract path as well as the results path within the run.py.
* Run the project using the command “python run.py”.
* After running the project, a text file would be generated within the results path provided with information.

****

**Customers and Users**

* Reached out to Shovon for clarification of Manticore Features
  + The team contacted Shovon for assistance to help clarify the features of Manticore. Shovon advised the team that Manticore is a very complex tool and has many features. He advised the team to focus only on the core functions of Manticore instead of all the features as a whole. The team is currency working now to identify which core features to add to Gas Gauge to create the base frame for GasCore+
* Reached out to TA Mohammad for documentation about Gas Gauge
  + The team contacted TA Mohammad for assistance in regards to finding documentation for Gas Gauge. The team has also requested clarification for some of Gas Gauge features from the TA in order to properly merge in Manticore features without interrupting the Gas Gauge features already in place in the GasCore+ tool.
* Reached out to Shovon for an update on Gas Gauge/Manticore specific Smart Contracts.
  + The team contacted Shovon for an update for finding Gas Gauge/Manticore specific smart contracts previously requested from the last iteration. Shovon is still currently looking for those contracts and instead of waiting for the smart contract, the team decided to create our own smart contracts that will be compatible with both Manticore and Gas Gauge and test the features of both tools as the team implements more features.

**References**

[1] [Slither: The Leading Static Analyzer for Smart Contracts | Trail of Bits Blog](https://blog.trailofbits.com/2019/05/27/slither-the-leading-static-analyzer-for-smart-contracts/) Accessed 10/2022

[2] <https://github.com/trailofbits/manticore/wiki/Tutorial:-Running-under-Manticore>, Accessed 13/22

[3] https://manticore.readthedocs.io/en/latest/evm.html?highlight=create\_account, Accessed 13/22

[4] <https://arxiv.org/pdf/2112.14771.pdf>, Accessed 15/22

[5] <https://buildmedia.readthedocs.org/media/pdf/manticore/latest/manticore.pdf> . Accessed 10.2022