Security Analysis Methods for Detection and Repair of DoS Vulnerabilities in Smart Contracts

by

Behkish Nassirzadeh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Computer Engineering

Waterloo, Ontario, Canada, 2021

© Behkish Nassirzadeh 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In recent years we have witnessed a dramatic increase in the applications of blockchain and smart contracts in a variety of contexts, including supply-chain, decentralized finance, and international money transfers. However, a critical stumbling block to their further adoption is smart contract security (or more precisely, the lack thereof). Smart contracts, once deployed on a blockchain, are immutable. Hence, unlike traditional software systems, smart contracts are particularly vulnerable to latent security issues. It is therefore imperative that security analysis tools be developed that help improve smart contract security if they are to have continued adoption and impact. A particularly widespread class of security vulnerabilities that afflicts Ethereum smart contracts is the gas-based denial of service (DoS). Briefly, these vulnerabilities generally present in contracts containing unbounded loops.

To address the described problem, we present Gas Gauge, a tool aimed at detecting gas-based DoS vulnerabilities in Ethereum smart contracts. Gas Gauge consists of three major components: the Detection Phase, Identification Phase, and Correction Phase. First, we describe a highly accurate static analysis approach that finds all the loops in a smart contract (the Detection Phase). Then, a set of inputs that causes the contract to run out of gas is generated using a fuzzing approach. The last component uses static analysis and run-time verification to predict the maximum loop bounds consistent with allowable gas usage automatically. This component uses a binary search approach and an independent parallel processing design to speed up the process. Each part of the tool can be used separately for different purposes or all together to detect, identify and help repair the contracts vulnerable to out-of-gas behaviors.

Gas Gauge was tested on 2,000 real-world solidity smart contracts. The results were compared to seven state-of-the-art tools, and it was empirically demonstrated that Gas Gauge is highly effective and useful.

Acknowledgements

I want to express my gratitude and appreciation to the following people for their help and support throughout my Master of Applied Science Degree.

First and foremost, I would like to acknowledge my supervisor Prof. Vijay Ganesh for his ongoing assistance, support, and guidance during my Master of Applied Science Degree. His mentorship helped me enormously to achieve many of my goals, including writing this thesis.

Special thanks to Dr. Sebastian Banescu for his continued advice and involvement in writing this thesis and throughout my Master of Applied Science Degree.

I would like to thank Dr. Albert Heinle for his advice and assistance. His guidance and feedback helped me significantly during my Master of Applied Science Degree.

I would like to thank Prof. Anwar Hasan for his support during my Master of Applied Science Degree and serving as one of my Master of Applied Science thesis reading committee members.

I would like to thank Prof. Lin Tan for serving as one of my Master of Applied Science thesis reading committee members.

I would also like to thank Huaiying Sun for her continued assistance in the project that led to this thesis.

Finally, I would like to thank my parents, sister, and friends for always being there for me. I could not have completed this thesis without their support.

Dedication

This is dedicated to my beloved family.

"To us, family means putting your arms around each other and being there."
-Barbara Bush

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smart contracts are one of the main features of blockchain. They are executable programs that allow building a programmable value exchange between various parties without a trusted third-party. Ethereum is a leading platform for smart contracts. Users can use the Ethereum network to make new contracts, invoke accessible functions of a contract, and transfer the cryptocurrency Ether (ETH) to other contracts or users. However, smart contracts are vulnerable to several security attacks.

Every transaction and operation in Ethereum smart contracts costs a certain amount of gas, a measurement unit for the amount of computational effort on the Ethereum Virtual Machine (EVM), paid by the transaction initiation party. As part of the gas mechanism on Ethereum, each transaction in a smart contract has an upper bound on the amount of gas that can be spent. This gas, which is called the Block Gas Limit, determines the amount of computation that can be done in a block of code. The sum of all transactions in a block cannot exceed this gas limit. If the execution gas usage surpasses this limit, the transaction fails [1]. As a result, the Ethereum Virtual Machine (EVM) raises an out-of-gas exception, and the transaction gets denied. The execution of a function in a vulnerable smart contract can be blocked indefinitely if out-of-gas conditions are not appropriately handled. Therefore, denial-of-service (DoS) attacks can occur to a vulnerable contract to block a transaction. This vulnerability is also known as *DoS with Block Gas Limit*[1].

Meanwhile, manual security analysis does not scale with the size and number of available smart contracts. This is the motivation behind smart contract verification tools. Many developers have implemented multiple security scanners using techniques such as static analysis, symbolic execution, or fuzzing in the past few years to audit smart contracts for security vulnerabilities automatically. However, many of these scanners produce

many false-positives or false-negatives. By verifying a smart contract against known vulnerabilities, the chances of deploying an exploitable contract are decreased. This is why it is essential to determine just how useful current smart contract verification tools are.

We conducted a survey and compared three types of smart contract verification tools: symbolic execution, static analysis, and a combination of the two. In this survey, we scanned a benchmark of 25 contracts using seven state-of-the-art smart contract verification tools and summarized the results by each tool's performance. While the surveyed tools have varying strengths and weaknesses, we noticed that none of the tools performed reliably in identifying DoS with Block Gas Limit vulnerability. It was concluded that identifying this vulnerability can be challenging for the current analyzing tools. Even if a tool can detect this vulnerability, it may not provide an exact point when the contract reaches the gas limit. Developers may need to spend hours to find this vulnerability, as it generally occurs in contracts with unbounded loops. Furthermore, finding the maximum loop bound before the contract runs out of gas can be challenging.

1.1 Problem Statement and Solution

1.1.1 Problem Statement

A noticeable amount of Ether is always used in smart contracts. Therefore, smart contracts are an engaging target for attackers. This puts more pressure on the developers to ensure the contract is free of bugs and vulnerabilities before deployment. Even though the remaining balance of a vulnerable contract to gas-based DoS attacks can be locked forever, most automatic scanners cannot detect all smart contract vulnerabilities reliably.

What makes these attacks unique is that they can be prevented by appropriate conditions or require statements. Generally, to avoid these attacks, one needs to determine the loops that can lead to a DoS attack, perform gas analysis to determine the exact point (i.e., loop iteration) when out-of-gas occurs. Then, one can add proper conditions or require statements to prevent waste of gas or add an additional sentinel parameter to the function with a loop allowing the logic to be broken into multiple transactions. However, the problem is that although detection of gas concerns due to loops is easy, identifying only the loops that can cause a contract to run out of gas is difficult. More importantly, finding the exact point where out-of-gas starts to occur is very challenging. This is because manual processing is potentially lengthy and a tedious task.

1.1.2 Solution

In this thesis, we address these challenges by designing and implementing an automatic tool called Gas Gauge. This tool contains three major algorithms. The first part is a static analysis method that efficiently and accurately detects all the loops in a smart contract. We refer to this part as the Detection Phase of the tool. The second part is a whitebox fuzzer based on static analysis and run-time analysis. It identifies public functions containing at least one input variable that affects a loop inside the function. The reason is that unbounded loops are the leading cause of DoS with Block Gas Limit. Next, it finds a set of inputs to the function that causes the contract to go out of gas by running and fuzzing it. This part is referred to as the Identification Phase of the tool. This last algorithm uses static analysis and run-time verification to predict the maximum allowed loop bounds in a contract. It uses a binary search approach and an independent parallel processing design to speed up the process. This part automatically infers the maximum number of allowed iterations of loops in a contract before it runs out of gas. The maximum number of allowed iterations is the threshold of that loop. The generated thresholds are given as a formula based on the gas limit of the contract, the gas usage in the first iteration, and the average gas usage of the other iterations of the loop. If the contract contains nested loops, the formula is also based on the thresholds of the outer and inner loops. This formula can be used in a "require" statement with the loop's bound to prevent out-of-gas errors. Hence, we refer to this part as the Correction Phase of the tool. The tool can conveniently detect, identify and help correct the contracts vulnerable to out-of-gas behaviors.

1.2 Contributions

This thesis makes four major contributions.

1. Design and implementation of a static analysis technique to efficiently and accurately find all the loops in a smart contract
2. Design and implementation of a static and run time analysis (hybrid) whitebox Fuzzer to find an out-of-gas instance in a smart contract containing public functions with loops
3. Design and implementation of a static analysis and run-time verification binary search-based method to identify loop thresholds in a smart contract

4. Verifying the result of the tool containing all the above methods and comparing them with seven state-of-the-art tools on 2,000 real-world smart contracts

1.3 Applications of Gas Gauge

Gas Gauge has an extensive range of applications for contract developers, testers, auditors, owners, and users. It can be used for gas-related vulnerability detection, debugging, verification, and certification of an Ethereum smart contract. Each component can be used separately for different applications or all together to detect, identify and help repair the contracts vulnerable to out-of-gas behaviors. To the best of our knowledge, Gas Gauge is the first tool that provides such information.

The Detection Phase of the tool uses static analysis to extract important information about contract's loops using components like Control Flow Graph (CFG), function signatures, variable dependencies, and so on and hence is fast and extremely accurate. It can be used to detect all the loops in a smart contract. It can also be a part of another tool if the tool requires an efficient and accurate loop finder.

The Identification Phase of the tool is a whitebox fuzzer based on static analysis and run-time analysis. It helps the users identify the blocks of code vulnerable to DoS with Block Gas Limit and provides them with inputs that can help the testing and debugging process. If the tool does not find such a block, it means the contract is most likely not at risk of this vulnerability at that block of code.

Finally, the Correction Phase ensures that if the loop bound is less than the reported threshold, the contract is free of out-of-gas vulnerabilities in that block of code. It also helps developers debug and improve their code if there is a chance for gas-related vulnerabilities. Since this part uses run-time verification, it has no false positives.

1.4 Summary of the Results

We argue that our experimental evaluation reveals that Gas Gauge is extremely useful and accurate. We have tested our tool on 2,000 real-world smart contracts extracted from etherscan.io [2], one of the most popular Ethereum blockchain explorers. All these contracts are manually scanned and made sure half of them contain at least one loop, and the other half doesn't contain any loops. Then, we compared our results against seven state-of-the-are tools and generated promising results. Our tool uses run-time analysis to ensure no false-positives, and our enhanced static analysis method ensures a low false-negative rate.

1.5 Thesis Organization

The rest of this thesis is organized in the following way. Chapter 2 provides the information needed for the rest of this thesis, briefly introduces the helper tools utilized to implement Gas Gauge, and describes the minor contribution of this thesis, which is a short survey of smart contract analysis tools. Chapter 3 describes the design and implementation of Gas Gauge components. Chapter 4 presents the setup, results of our experimental evaluations and the limitations of Gas Gaugeas well as the related work and tools to Gas Gauge. Finally, the conclusion and future work are presented in Chapter 5.

Chapter 2

Background

2.1 Background Information

Smart contracts are an application of blockchain and meant to store rules and agreements made by several parties and execute an agreement when they are triggered. Many industries are using smart contracts to perform trustless computations. Ethereum is one of the leading blockchain platforms.It is a decentralized and open-source blockchain that contains millions of accounts and billions of USD in capitalization, and hence, one of the most prevalent underlying technologies for smart contracts [3]. They are commonly written in the Solidity language, which is a Turing-complete JavaScript-like scripting language. Solidity is tailored to smart contract-specific constructs, such as transferring funds, interacting with and creating new smart contracts, and querying state variables on the chain. it then gets compiled to Ethereum Virtual Machine (EVM) assembly instructions to be deployed on the blockchain.

Smart contracts handle transactions in a cryptocurrency called Ether. Transactions and procedures in Ethereum smart contracts require a certain amount of gas, a measurement unit for the amount of computational effort on the Ethereum Virtual Machine (EVM). This gas has to be paid by the transaction initiation participant. Thus, The more computational or storage resources required to execute a transaction, the more gas needed. Gas-related vulnerabilities are one of the most critical kinds of smart contract vulnerabilities.

Each transaction or function execution in smart contracts is bounded to a certain amount of gas known as the Block Gas Limit. If the execution gas usage exceeds this limit, the transaction gets reverted, and an out-of-gas error occurs [1]. If the out-of-gas error

is not handled properly, the execution of that function can be blocked indefinitely. As a result, denial-of-service (DoS) attacks can target contracts with gas-related vulnerabilities. One of the principal kinds of gas-based vulnerabilities is known as DoS with Block Gas Limit vulnerability, which mainly occurs in contracts with unbounded loops. Even harmless programming patterns in centralized applications can lead to Denial of Service conditions in smart contracts. This can happen when the cost of executing a function exceeds the block gas limit [4].

2.2 Whitebox Fuzzing

Nowadays, security vulnerabilities in software products can be found by two fundamental methods: Code inspection of binaries and blackbox fuzzing. Blackbox fuzzing is a class of blackbox random testing that randomly mutates well-formed inputs to the program and then tests the program on the resulting data in order to trigger a bug [5]. Blackbox fuzzing is an effective method to test a program; however, it can have some limitations. Low code coverage is one of the leading limitations of blackbox fuzzing resulting in missing security bugs [5]. An alternative approach to blackbox fuzzing is whitebox fuzzing. It is a type of automatic dynamic test generation, based on symbolic execution and constraint solving, intended for large applications' security testing [5].

2.3 Helper Tools

Developing an automatic gas analysis tool from Solidity source code requires a considerable implementation effort that has been possible thanks to many existing open-source tools. Slither [6] and Truffle Suite [7] are used to implement Gas Gauge. In particular, Slither is used in the static analysis part of the implementation, and Truffle is used in the run time analysis part. Slither provides many useful APIs to collect information, such as data dependencies and function signatures. This is used to summarize the contract and extract needed information for the other parts of the implementation. Slither also produces the Control Flow Graph (CFG) of the contract. The CFG is used to find the loops in the contract and their orders. Moreover, Truffle Suit is used to compile and deploy Solidity smart contracts to a test Ethereum network. This allows us to use different sets of inputs in the Identification Phase of the tool and use different threshold values in the Correction Phase while retrieving useful gas-related information such as gas used and gas left.

2.3.1 Slither

Slither [6] is a static analysis-based tool for examining smart contracts. Based on their paper [6], Slither takes as initial input the Solidity Abstract Syntax Tree (AST) of a smart contract generated by the Solidity compiler from the contract's source code. Firstly, Slither recovers some critical information from the input Solidity AST such as the contract's inheritance graph, the Control Flow Graph (CFG), and the list of expressions. Then, it transforms the entire code to SlithIR, which is an internal representation language. SlithIR facilitates the computation of a variety of code analyses using Static Single Assignment. Generally, Static Single Assignment is a common representation in the compilation and static analysis [6]. During the third stage, it does a set of predefined analyses such as the reads and writes of variables analysis, the protected functions analysis, and the data dependency analysis. This stage provides enhanced information to the other modules, such as the vulnerability detection module for bug finding, the optimization detection module for gas optimization, and the printer module for displaying the contracts' information, or for customized third-party tools.

2.3.2 Truffle Suite

In many parts of Gas Gauge, we used Truffle and Ganache that are the tools available in Truffle Suite [7]. Truffle is a world-class and powerful development environment for blockchains using the Ethereum Virtual Machine (EVM). Truffle offers built-in smart contract compilation, linking, deployment, and binary management. It also offers automated contract testing for fast development [7]. Truffle allows developers to write manageable and straightforward test files for smart contracts in two different ways:

1. In JavaScript and TypeScript, for exercising your contracts from the outside world, just like your application.

2. In Solidity, for exercising your contracts in advanced, bare-to-the-metal scenarios.

Ganache is a personal blockchain for rapid Ethereum. One can use it across the entire development cycle to develop, deploy, and test dApps in a safe and deterministic environment [7].

2.4 A Short Survey of Security Analysis Methods

Because smart contracts' security vulnerabilities have caused users to lose millions of dollars, many tools have been developed to scan the contracts and identify potentially vulnerable code blocks. Standard techniques in these tools are static analysis, symbolic execution, and fuzzing. We picked seven popular smart contract verification tools to put them to the test. The selected tools are the industry standard tools that use static analysis, symbolic execution, or a combination of both. The reason that fuzzers were not part of this comparison is that they generally require manual test cases. It means the process is time and resource consuming, and the quality of the results is affected by the quality of the test cases made. The seven chosen tools are Mythril [8] , Securify [9], Securify2 [10], Smartcheck [11], Slither [6], Manticore [12], and MPro [3]. In this survey, we used these tools to scan a benchmark of 25 contracts containing 31 different kinds of vulnerabilities and warnings. We evaluated the tools by each tool's performance in terms of identified vulnerabilities, the number of contracts scanned, false-positive rate, and false-negative rate. The standard for correctness was determined by the manual auditing of smart contracts performed by Quantstamp Certifications [13], the Smart Contract Weakness Classification Registry [4], and [14]. We also manually audited some of the contracts to ensure the correctness of the results. This survey was conducted between January to February 2020, and the most updated version of each tool at that time was used.

2.4.1 The Main Goals of the Survey

The main goals of this survey were as follows:

1. A comparison between different common methods used in general smart contract verification tools
2. A comparison between some of the well-known general smart contract verification tools
3. Identifying critical vulnerabilities that are challenging to be detected by the well-known tools

2.4.2 The Results of the Survey

The result of our survey is summarized in Table 2.1.

Table 2.1: Smart Contract Verification Tools Comparison

Tool Name	**Method**	**Number of Contracts Successfully Scanned**	**False Negative Rate (%)**	**Identified Vulnerabilities**
Securify2.0	Static Analysis	10	53	DoS with Failed Call Unlocked Pragma
Slither	Static Analysis	14	56	Unexpected Ether balance Unlocked Pragma Weak Randomness
SmartCheck	Static Analysis	**25**	**48**	**DOS With Block Gas Limit** DoS with Failed Call Integer Over/underflow Unexpected Ether balance Unlocked Pragma Weak Randomness
Mythril	Symbolic Execution	12	84	DoS with Failed Call Integer Over/underflow Unexpected Ether balance
Manticore	Symbolic Execution	5	80	Unexpected Ether balance
Securify	Symbolic Execution	16	71	DoS with Failed Call Integer Over/underflow Reordering attack Unexpected Ether balance
MPro	Static Analysis and Symbolic Execution	21	93	Integer Over/underflow Unexpected Ether balance Weak Randomness

Tools may not be able to scan a contract due to many reasons, including the unsupported Pragma version of the contract, reaching the default time out, and so on. As we can see in this table, SmartCheck is the only tool that scanned all the contracts. The false-positive rate is calculated as the number of false-positives divided by the total number of negatives (false-positive + true-negative). Each contract was only audited for particular vulnerabilities. If a tool reported a vulnerability that was not checked by manual auditing, the vulnerability was not counted towards either true or false positive. Hence, the false-positive rates of all the tools are meager and not shown in the table. However, in general, we expect some false-positives for static analysis tools and zero for symbolic execution tools. The false-negative rate is calculated as the number of false-negatives divided by the total number of positives (true-positives + false-negatives). As shown, SmartCheck performed the best in this category as well. The Identified Vulnerabilities column lists the vulnerabilities identified correctly (true-positive) by each tool in at least one of the contracts in the benchmark. Smartcheck was able to identify the most number of vulnerabilities and hence the winner of this category.

Static analysis tools have better results overall, especially in terms of scan rate and false-positive rate. Also, Smartcheck outperformed other tools in most categories, and hence is the winner of the comparison. However, we discovered that although gas-related vulnerabilities are incredibly critical, many available tools cannot identify them reliably. In our survey, most tools had difficulties identifying DoS with Block Gas Limit, one of the principal gas-related vulnerabilities. Three of the contracts in our benchmark were at risk of this vulnerability, and only Smartcheck was able to identify it in only one of the contracts. As a result, we designed and implemented Gas Gauge dedicated to this vulnerability. The next chapter explains the structure of Gas Gauge.

Chapter 3

Description of Gas Gauge Components

Gas Gauge was designed and implemented to address the gas-based vulnerabilities of smart contracts. Since loops are the main cause of many gas-related vulnerabilities, the focus of Gas Gauge is on loops. The main methods used in Gas Gauge are static analysis, run-time analysis, and whitebox fuzzing, and it has many different components.

Gas Gauge contains three major parts: the Detection Phase, Identification Phase, and Correction Phase. The user has the option to use each of the parts separately or use them all together by passing in appropriate booleans. The Identification Phase and Correction Phase require information generated by the Detection Phase; however, Identification Phase and Correction Phase are independent of one another. The inputs to all methods are the Solidity source code and the contract's gas limit (optional if only Detection Phase is used).

The Detection Phase uses an advanced static analysis approach to efficiently and accurately detect all the loops in a smart contract. Slither [6] is utilized to enhance this part. The Identification Phase is a whitebox fuzzer based on static analysis and run-time analysis. It identifies the public functions containing at least one input variable that affects a loop inside the function. It uses static analysis to find the proper functions and inputs to fuzz and uses a run-time white fuzzing technique to find a set of inputs to the function that causes the contract to go out of gas. The Correction Phase uses static analysis and run-time verification to predict loop thresholds. It is enhanced with a binary search approach and an independent parallel processing design to speed up the process. This part identifies all the contract's functions containing loops using static analysis. Then, it automatically infers the maximum number of allowed iterations before the contract runs out of gas. It

does so by using run-time verification and is improved by a binary search approach. The maximum number of allowed iterations in a loop is referred to as the threshold of that loop. The generated thresholds are given as a formula based on the gas limit of the contract, the gas usage in the first iteration, and the average gas usage of the other iterations of the loop. If the contract contains nested loops, the threshold also has a formula based on the outer and inner loops. This formula can be used in a "require" statement or any other conditional statement with the loop's bound to prevent out-of-gas errors. Therefore, it can be used to repair (correct) contracts with gas-related vulnerabilities.

The structure of Gas Gauge is as shown in Figure 3.1. The following sections describe each component of Gas Gauge in detail.

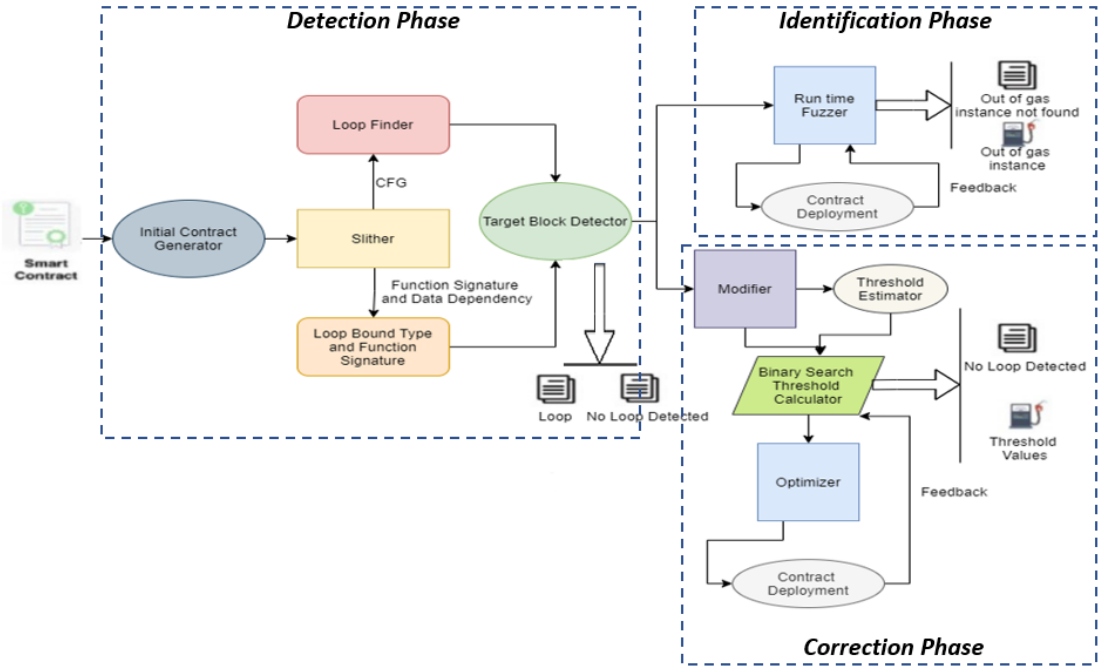

Figure 3.1: The Structure of Gas Gauge

3.1 Detection Phase

3.1.1 Initial Contract Generator

The first and the simplest component is the Initial Contract Generator. In this step, a copy of the original contract is made. If the Correction Phase is needed, the copied version is formatted to make it easier for the other parts. For example, it removes all the comments, and adds brackets and spaces if needed. Then, the new contract is fed to Slither.

3.1.2 Target Block Detector and Its Inputs

The Target Block Detector is one of the most important components of the design. In this stage, Slither and other static analysis approaches are used to detect loops in the contract. It does so by obtaining the CFG of the contract and looking for any loops in the CFG. Also, the visibility of the functions containing loops and their signature is extracted. If only the Detection Phase is needed, the program halts here and outputs the functions containing loops along with the number of loops in each function. In any case, if the contract does not contain any loops, the program halts here and outputs "No loop detected".

If the Identification Phase or Correction Phase is also needed, the types of variables affecting the loop bounds are obtained. This process involves an advanced static analysis and the printers available in Slither to gather the loop conditions, the variables affecting the loops, variable dependencies, and function summaries. The variables affecting the loop bounds are classified into four groups: State, Local, Constant, and Input variables. State variables are essentially the contract variables whose values are permanently stored in contract storage and can be accessed by all the contract functions. Constant variables are the ones that only carry a fixed value and are defined within the target function. Input variables carry their usual meaning: the inputs of the target function. Finally, Local variables are declared and initialized inside the target function, and their context is within a function and cannot be accessed outside. If a Local variable is detected, the Target Block Detector performs induction to find a list of all State, Input, and Constant variables affecting that local variable.

For the whitebox fuzzer, we can only fuzz public functions, and the aim is to find a set of inputs that affect the loop bounds. Therefore, the Target Block Detector finds public functions that contain loops that their bounds are affected by input variables (Input variables or Local variables affected by Input variables) and passes them to the Identification Phase. If the Target Block Detector does not find a function satisfying mentioned criteria

, the program halts. After identifying the target functions, the function signatures, and the name, and the type of the input variables affecting the target loops are passed to the run-time fuzzer.

For the Correction Phase the following is gathered and passed to the next part.

1. The function signatures for all the functions containing loops
2. Information about the loops, including the scope of each loop, and the type of each loop (for, while do-while)
3. A list containing the order of the loops if the function contains nested loops
4. Information of the variables affecting the loop bounds and their types

3.2 Identification Phase

In the Identification Phase, a whitebox fuzzing approach is utilized to generate a set of inputs for each unbounded loop in a public function in an Ethereum smart contract. The bounds of these loops must be affected by at least one of the input variables of the function containing the loop; otherwise, fuzzing cannot be effective. This component takes information from the previous parts of the tool and the contract's gas limit as inputs and outputs the set of values that make the contract go out of gas. If the contract does not have any loops or the fuzzer cannot find the correct inputs after a predetermined number of tries, it halts. Here, by public functions, we mean functions that can be called from the outside of the contract. Thus, only functions with "public" or "external" visibility can be fuzzed using our tool. The difference between the two types of visibilities is that "public" functions can be called internally or via messages, while "external" functions can only be called from other contracts or via transactions and cannot be called internally [15]. On the other hand, "internal" functions can only be accessed internally, meaning from within the current contract or contracts deriving from it, and "private" functions are only visible for the contract they are defined in [15]. Hence "private" and "internal" functions cannot be fuzzed directly without modifying the contract.

In this step, all the input variables in the target function get set to their initial values (i.e., integers are set to zero, and arrays are set to an empty array). The input values reported by the Target Block Detector get encoded in their binary representations. For instance, for an integer variable, the initial state is 256 zeros. Then the tool picks a bit at

random and flips it. Thus, if the value of the chosen bit is 0, it gets changed to 1 and 0 otherwise. Then, the binary encoding gets converted back to the original form. The only exception is arrays. In this case, mainly the array size can affect the bound of a loop, and hence, arrays get represented by 256 bits as arrays in Solidity can have up to 2^{256} elements. Instead of converting the binary representation to the original array form, it gets converted to an integer, and the array size gets set to that. If multiple input variables affect the loop bound, the binary representations get concatenated, the bit is flipped, and the concatenated value gets converted back to the original forms.

Next, all the necessary files and test cases are generated automatically, and different inputs are tested until the desired set of values is found. The goal is to find a set of inputs that makes the contract run out of gas. Truffle Suite [7] is used to deploy the contract to a test Ethereum network. This suite has a tool called Ganache, which provides a test Ethereum network, and another tool called Truffle, which compiles and deploys Solidity smart contracts to this network. A test file is generated in Solidity. This test file is used to call the target function with generated input values. After generating all the needed files and information, the contract gets deployed, and the target function gets fuzzed (tested with different input values). Suppose the test case halts and returns the remaining gas in the contract, the process repeats, and the fuzzer flips another bit. The process continues until a set of input is found that makes the test case abort due to an out-of-gas exception. At this point, the set of inputs used is reported as the output of the tool. To ensure the tool does not run forever, users can choose the maximum number of bit flips.

Once the fuzzer finds set of input that cause the contract to run out of gas, it prints out the function, the input variables affecting the loop, and the values that caused an out-of-gas error for each of these variables. If for any reason, the tool is not able to find the set before a halting condition is met, it only prints out the function signatures and the information about input variables that might affect the loop bounds.

3.3 Correction Phase

The Identification Phase is particularly useful for developers or contract owners to scan the contracts before deployment and check if the contract is at risk of DoS with Block Gas Limit. Since the tool also provides an instance, it can help them examine the problem further. However, one of the first steps to fix the code is to find the exact point at which the out-of-gas condition starts to get triggered, and any arbitrary set of inputs is not enough. This is the motivation behind the Correction Phase, which is intended to find the upper bound limit of the loops in a smart contract.

The output is a formula based on the maximum number of allowed iterations for loops before the contract runs out of gas. We refer to this number as the threshold of the loop. If the tool cannot find a threshold, it provides information such as the type of the loop and the variables affecting its bound.

This part has multiple components. A detailed explanation of each of these components is provided in the next subsections.

3.3.1 Modifier

The Modifier makes two copies of the contract generated in the Initial Contract Generator and modifies them separately for each identified loop. The first is to measure the gas used for the first and second iteration of each loop identified by the Target Block Detector. It is because, generally, the first iteration of each loop consumes more gas than the other iterations. The second iteration's gas usage is typically the average gas usage of all the other iterations of the loop. The reason is that in most loops, a similar operation happens in each iteration. The first set is the input to the Estimator. The second is to change the loop bound to the desired value. It allows us to run each loop with a specified number of iterations and capture the gas left after that many iterations. The binary search model uses this to test different values. The modifications only have an insignificant effect on the behavior and gas usage of the contract. For both modified copies, a public function is added as a wrapper to call the target function. This function can call "public", "private", and "internal" functions but not "external" ones. Therefore, external functions are changed to "public", so they can be called as well.

3.3.2 Threshold Estimator

The Threshold Estimator receives the first modified contract and automatically creates a solidity test file and all the other necessary files for Truffle Suite [7]. Also, to get an actual gas usage for each iteration, the new contract and test file are deployed to a test Ethereum network. The Truffle Suite is used to deploy the contracts. The modified contract runs each loop twice and captures the gas used in each iteration. One can call the function "gasleft() returns (uint256)" to get the gas left at any instance of the contract. This function always exists in the global namespace and returns the remaining gas at any instance [16]. We can call gasleft(), the target function or block of code in our contract, call gasleft() again, and get the difference between the returned values of the two gasleft() calls. This way, we can accurately and efficiently calculate the gas usage of the target block. Next, based

on the gas usage amount reported by the Truffle Suite and the gas limit, the maximum number of iterations that the loop can perform without running out of gas is estimated. The pseudocode is shown as Algorithm 1.

Algorithm 1: Estimated threshold for a loop in a function

```
1 initial_gas = gasleft() ;
2 while iteration < 2 do
3 |   original code inside the loop ;
4 |   if iteration == 0 then
5 |   |   gas_1 = initial_gas − gasleft();
6 |   end
7 |   gas_2 = initial_gas − (gas_1 + gasleft());
8 end
9 max_iteration = 1 + (initial_gas − gas_1)/gas_2 ;
```

3.3.3 Binary Search Threshold Calculator

With the gas mechanism on Ethereum, any instruction's execution needs a relatively steady amount of gas. Hence, we first use the run-time/static analyzer to get the estimated threshold for a loop bound. It means any value over this threshold may trigger the out-of-gas condition. To some degree, the estimated threshold is over or underestimated. One reason is that there might be more complicated code inside the loop that varies by the iterations. Also, the gas usage of other parts outside the loop is ignored in this way. Therefore, we used a run-time verification approach to find a more accurate value. We further use a binary search approach to cut down the action space rapidly. The estimated threshold helps the binary search model have a proper starting point.

The second set of the modified files and the Estimator's result are fed to the Binary Search Threshold Calculator. The purpose is to run each loop with a specified number of iterations and capture the gas left after that many iterations. The binary search model uses this to test different values. In order to explain how the contract is modified, an example is given below.

The code snippet below shows the original contract that contains two unbounded loops.

```
pragma solidity >=0.4.24 <0.7;
contract TestContract{
    uint[] numbers;
    function addNumbers(uint[] calldata newNumbers) external
       returns(uint){
        for(uint i=0; i<newNumbers.length;i++) {
            numbers.push(newNumbers[i]);
        }
        uint sum = 0;
        for(uint j=0; j<numbers.length; j++){
            sum += numbers[j];
        }
    }
}
```

The code snippet below illustrates how the contract is modified in order to find the threshold for the first loop.

```
pragma solidity >=0.4.24 <0.7;
contract TestContract {
    uint[] numbers;
    function getStateVarInaddNumbers(uint[] memory input1) public
       returns (uint) {
        addNumbers(input1);
        return gasleft();
    }
    function addNumbers(uint[] memory newNumbers) public returns
       (uint){
        uint loopCounter = 0;
        for(uint i=0; loopCounter < VALUE; i++) {
            numbers.push(newNumbers[i]);
            loopCounter = loopCounter + 1;
        }
        uint sum = 0;
        for(uint j=0; 1 == 0 ; j++){
            sum += numbers[j];
        }
    }
}
```

As shown, a wrapper function is added, the visibility of the target function is changed from "external" to "public", the second loop's condition is modified, so it does not get executed, and the condition of the target loop is changed to execute the loop up to a desired number of iterations. $VALUE$ is the number generated by the Binary Search Threshold Calculator. Also, if the original loop bound is affected by an input variable as identified in Target Block Detector, the generated test file is modified, so the value or size of the input variable in the wrapper function is set to $VALUE$ where applicable. This ensures the function does not fail if there is a condition that requires a specific value or size of that variable inside the function. For example, in the shown code snippet, "input1" is set to an array of integer of size $VALUE$. Similarly, if a state variable affects the loop bound, the value or size of the variable is set to $VALUE$ inside the wrapper function before calling the target function. For example, to test the second loop in the given code snippet, "numbers" is set to an array of integers of size $VALUE$ before calling "addNumbers".

At this stage, by inputting different values to the target function, we can calculate different gas usages. The goal here is to find two consecutive numbers, where only the larger number makes the contract run out of gas. As mentioned before, the original contract and test file are modified to change the loop bound to the desired value. Each loop in each function is isolated to ensure the value is only affecting the target loop. For each value, the contract is deployed and ran and the gas left is obtained. There is a lower bound and an upper bound limit for the search space in the Binary Search model. They are initially set to 0 and 5,000, respectively. Truffle has a time limit that one can run a contract. Once the execution of a contract passes this limit, it throws a time out exception and stop running. Therefore, if a loop runs for too many iterations, it might trigger the time out exception. From our experiments, if the maximum number of loop iterations is less than 5,000, the test does not trigger the time out condition. Also, if a loop threshold is over 5,000, there is a lower chance of running out of gas as most contracts do not require that many iterations in one loop. Hence, the upper bound is set to 5,000. The estimated threshold is used as the starting point of the binary search. If the contract still has gas after testing a new value, the original $value + 1$ and otherwise the original $value - 1$ is used as the next loop bound. If the contract still has gas, the lower bound is changed to the $value + 1$; otherwise, the upper bound is changed to $value - 1$. Then, the new test value becomes the midpoint of the lower and upper bound. The process repeats until the model finds two consecutive numbers that only the larger one consumes all the gas.

The algorithm for this part is shown in Algorithm 2. The termination conditions are as follows:

1. The loop threshold is found

2. The lower bound becomes larger than or equal to the upper bound

3. The gas usage does not change with the change of the loop bound

4. An error is encountered

Algorithm 2: Run-time threshold of a loop in a function

```
1  guess = estimated value by static and run-time analysis;
2  lower = 0;
3  upper = 5,000;
4  while not termination_condition do
5  |   gas_left =deploy the contract(guess);
6  |   if gas_left < 0 then
7  |   |   new_gas_left = deploy the contract(guess − 1) ;
8  |   |   if new_gas_left < 0 then
9  |   |   |   upper = guess − 1;
10 |   |   else
11 |   |   |   threshold = guess − 1;
12 |   |   |   return threshold;
13 |   |   end
14 |   else
15 |   |   new_gas_left = deploy the contract(guess + 1) ;
16 |   |   if new_gas_left > 0 then
17 |   |   |   lower = guess + 1;
18 |   |   else
19 |   |   |   threshold = guess + 1;
20 |   |   |   return threshold;
21 |   |   end
22 |   end
23 |   guess = upper + lower/2 ;
24 end
```

3.3.4 Optimizer

The last part is the Optimizer. It has two primary purposes. First, if the lower bound reaches 5,000 or the value reported by the Estimator is over 5,000, it calculates the approximate threshold and returns it as the output. It does so by running the contract and setting the target loop bound to 5,000. Based on the original gas, gas consumed by the

first iteration, gas consumed by 5,000 iterations, and the remaining gas, it estimates the loop's threshold. This is used when the binary search is not able to find the threshold or the threshold is larger than 5,000, and testing may take a long time or cause the test to reach the time out limit.

Furthermore, to test a value in the Binary Search Threshold Calculator, the system needs to run the contract with the target value, wait for the result, and then run the contract with either value +1 or value -1. Each execution takes a few seconds; hence running the contract twice each time may take a while. To speed up the process, the Optimizer makes two extra copies of the generated files. Then, it changes the files slightly to run the three contracts with three consecutive numbers simultaneously. These numbers are the original number, the number plus one, and minus one. Then, based on the feedback from running the original number, it decides to use the correct extra feedback. This way, the run-time of the binary search gets reduced by almost 50%.

3.3.5 Output of the Correction Phase

An example of the output of the Correction Phase is provided in Figure 3.2.

```
Report for 0xd848f9b61affaaa2e5a7402e87d27eaa0cc27b6f.sol:
Found 1 loop in 1 function with the following information:
Function Name: VAPE.multiTransfer(address[],uint256[]), Visibility: public, Number of Loops: 1
-----------------------------------------------------
Results for VAPE.multiTransfer(address[],uint256[]):
inputs affecting the function loop bound(s) are: input 1 (address[] memory receivers);

Results of the Threshold Finder:
for loop 1 in VAPE.multiTransfer(address[],uint256[]):
The type is Normal
For gas limit = 6721975, the calculated threshold = 899
gas consumption for the first iteration = 47420; average gas consumption for other iterations = 5932
Threshold formula (gasleft() is the global function gasleft() placed before entering the loop ) =
(gasleft() - 47420) / (5932)
-----------------------------------------------------
Run time = 422.0240927560001 seconds
```

Figure 3.2: Output of Correction Phase

The output contains the signature of the functions containing loops and the number of loops in them. It also has the variables affecting the loop bounds and their datatype. It provides the type of the loop (Normal/Single or Nested). Then, it provides the value of the threshold found by the tool for the provided gas limit along with the gas consumption of the first iteration and the average gas consumption for the other iterations. Finally, the threshold formula is given in the following format:

$$(gasleft() - g_1)/(g_2)$$

If there are nested loops in the contract, the formula is slightly different. The formula for the inner loops is similar to the ones above, but it considers the outer loop has only one iteration, and for the outer loops is similar to the following:

$$(gasleft() - g_1)/(g_2 + Internal)$$

Where $gasleft()$ is the value returned by the function gasleft() placed right before the loop in the source code, g_1 is the gas consumption of the first iteration of the loop, g_2 is average gas consumption of the other iterations, and $Internal$ is the gas consumption of the internal loops.

This formula can be used in a "require" statement to fix the code and prevent out-of-gas exception for that loop. The statement has to be placed in the source code and right before the loop. The user has to find the loop bound manually and place it in the "require" statement. So, an example of the "require" statement looks like this:

require($loopBound < (gasleft() - g_1)/(g_2)$, "Loop bound is over the threshold!");

For instance, for the given example in 3.2, the size of the input variable "receivers" is the bound of the loop, the "require" statement looks like this:

require($receivers.length < (gasleft() - 47420)/(5932)$, "Loop bound is over the threshold!");

In this case, if the user inputs an array of size greater than 899, the "require" statement gets triggered, and the execution stops before entering the loop.

3.3.6 The Time Complexity of the Correction Phase

In Solidity, a loop bound is an unsigned integer that can have a value from 0 to 2^{256}. If we assume this number is n for a given loop, the naive brute-force approach would require running the loop with up to n different loop bounds until finding the right threshold. Hence the time complexity of a naive brute-force approach is $O(n)$. We can improve this by using a binary search approach. We can cut the action space in half every time we use a new value as the loop bound in this approach. Hence, the time complexity of a binary search approach is $O(log(n))$. Due to the limitation of Truffle and to improve the run-time, we set the maximum number for n to 5,000. If the actual threshold is over 5,000, the risk of the contract running out of gas is low, and the threshold is only estimated based on the gas usage of the first and second iteration and the gas left after 5,000 iterations. Thus the loop only gets executed three times. If the actual threshold is under 5,000, the loop is executed at most 13 times ($\log_2(5,000)$). Therefore, the Correction Phase is extremely fast and efficient compared to other methods.

Chapter 4

Experimental Evaluation

4.1 Setup

We gathered a benchmark of 2,000 real-world Solidity smart contracts from etherscan.io [2]. These contracts were manually handpicked to ensure 1,000 of them contain at least one loop, and 1,000 of them do not contain any loops. Also, information such as the number of functions, loops, and lines of code was manually collected from the 1,000 contracts with loops in order to help us evaluate our results. Hence, in many cases, these values are used as the ground truth.

We ran our experiments on a machine that was equipped with 8GB of RAM, a 4-core Intel Xeon 2.2 GHz processor with Ubuntu 18.04 running. To test Gas Gauge, the latest version of nodejs, Ethereum, truffle, ganache-cli, solc-select, python, and Slither were installed. We deployed the target smart contract to a private chain using Ganache-cli.

Seven state-of-the-art tools were chosen to compare the results of Gas Gauge with them. These seven tools are a combination of Gastap [17] and Gasol [18], Madmax [19], MPro [3], Mythril [8], Securify2 [10], Slither [6], and SmartCheck [11].

MPro, Mythril, Securify2, Slither, and SmartCheck were installed using the instruction provided on their official websites or documentation. There is no access to the source code of Gastap and Gasol to test these tools; however, they have provided a web interface (https://costa.fdi.ucm.es/gastap) to try a combination of the two tools. Thus, this interface was used manually to scan our benchmark. MadMax is not open source either. However, it is built into Contract Library [20]. Therefore, we manually searched for the contracts in our benchmark and collected the reports generated by MadMax. As a result, in our analysis, MadMax and GasTap/Gasol do not have a run-time.

In all these experiments, Solc version 0.5.3 was used unless either the tool or the contract required a different version.

The following subsections describe our different evaluations and our findings.

4.2 The Detection Phase

4.2.1 The Experimental Evaluation

In the experiment, we scan the 2,000 smart contracts using the Detection Phase of Gas Gauge and seven industry-standard tools. This experiment's goals were to verify the results generated by our tool and compare them with well-known tools. The standard correctness was determined by the information such as the number of loops and the number of functions containing loops collected manually. The summary of this experiment is presented in Table 4.1 and 4.2. Table 4.1 contains the summary of the benchmark containing 1,000 contracts with loops, and Table 4.2 shows the summary of the benchmark containing 1,000 contracts without loops. The timeout limit for this experiment was set to 3 hours per contract for each of the tools.

Table 4.1: Detection Phase Comparison on 1,000 Smart Contracts with loops

Tool Name	**Method**	**Number of Contracts Successfully Scanned**	**False Negative Rate (%)**	**False Positive Rate (%)**	**Average Run-time (s)**
Gas Gauge	**Static Analysis**	**997**	**0**	**0**	**10**
GasTap+Gasol	Static Analysis and static reasoning	120	36	0	-
Madmax	Static analysis	921	79	0	-
MPro	Static Analysis Symbolic Execution	851	100	0	242
Mythril	Symbolic Execution	870	100	0	3109
Securify2	Static Analysis	548	47	0	176
Slither	Static Analysis	997	85	0	2.6
SmartCheck	Static Analysis	**1000**	47	0	**2**

It is important to mention that we could only obtain the reports of MadMax if the contract existed on Contract Library. Thus, the column representing "Number of Contracts Successfully Scanned" shows the number of contracts available on Contract Library. As a result, if a contract does not exist there, it is not necessarily a shortcoming of MadMax. Also, since the interface provided to test GasTap/Gasol required manual testing and did not support the majority of our contracts, this tool was ignored for the benchmark of contracts without loops.

Table 4.2: Detection Phase Comparison on 1,000 Smart Contracts without loops

Tool Name	**Method**	**Number of Contracts Successfully Scanned**	**False Positive Rate (%)**	**Average Run-time (s)**
Gas Gauge	**Static Analysis**	**1000**	**0**	**3.9**
Madmax	Static analysis	930	0	-
MPro	Static Analysis Symbolic Execution	980	0	167
Mythril	Symbolic Execution	949	0	1590
Securify2	Static Analysis	848	0	32.9
Slither	Static Analysis	**1000**	0	**1**
SmartCheck	Static Analysis	**1000**	0	1.8

Smartcheck is the only tool that successfully scanned all the contracts, and as shown, it was the fastest tool. Although Smartcheck was the fastest tool, its accuracy is much lower than Gas Gauge in detecting loops. When collecting information from SmartCheck, "SOLIDITY_GAS_LIMIT_IN_LOOPS" is the vulnerability we considered. GasTap has the least number of scanned contracts; however, it was able to detect loops in the majority of those contracts. For Mythril and MPro, we were looking for "SWC ID: 128" vulnerability; however, neither tool detected such vulnerability, and no gas-related or DoS attack vulnerability is reported in the list of supported vulnerabilities. The run-time of both tools was more than the others as they use symbolic execution at their cores. Furthermore, both Slither and Securify2 were able to detect some of the loops in a reasonable time. "calls-inside-a-loop" and "costly-operations-inside-a-loop" are the vulnerabilities checked for Slither, and "External call in loop" is the one checked for Securify2. MadMax, which is one the most common tools designed only for gas-based vulnerabilities, was only able to detect some of the functions containing loops. "DoS (Unbounded Operation)" and "DoS (Induction Variable Overflow)" are the kinds of warnings checked when summarizing the reports of MadMax.

The second table does not contain any column for false-negative rates as non of the contracts in that benchmark has any loops; thus, no false-negative can be reported. Also, An impressive result is that neither tool has any false-positives.

As we can see, Gas Gauge was able to detect all the loops in the contracts that it was able to scan. There were three contracts that Slither was not able to scan, and as a result, our tool was not successful in checking them as well. The reason is that the report generated by Slither is an essential part of our algorithms. Typically a tool may not scan a contract for various reasons like reaching the timeout, lack of support for the Pragma version, etc. Therefore, Gas Gauge can detect loops efficiently and accurately.

4.2.2 Performance Analysis

In this experiment, we tried to find the main factors affecting the run-time of the Detection Phase of Gas Gauge. We obtain the summary of each of the 1,000 contracts containing loops in our benchmark. This summary contains the total number of functions, number of functions containing loops, number of code lines (only source code lines without comments and blank lines), and number of loops in each contract. We also measured the run-time for each of the contracts for the Detection Phase. Figure 4.1 and 4.2 demonstrate our findings.

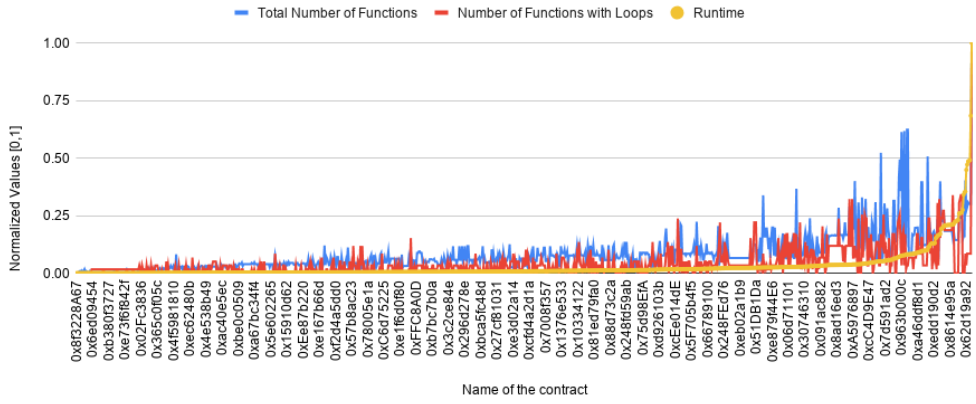

Figure 4.1: Effect of the Number of Functions on the Run-time

As we can see, the total number of functions and functions with loops have some impacts on the run-time. However, these effects are not as noticeable as the number of loops and lines of code. It appears that the order of the factors impacting the run-time is the number of lines of code, number of loops, number of functions containing loops, and total number of functions. This is also expected because Detection Phase uses a static analysis approach as its primary method.

ConsenSys has a tool called Solidity-metrics [21]. This tool provides Source Code Metrics, Complexity, and Risk profile reports for projects written in Solidity. One of the factors they provide is "Complexity Score", a custom complexity score derived from code statements known to introduce code complexity such as branches, loops, calls, and external interfaces. We obtained this score for each of the contracts and observed the impact on the run-time. Figure 4.3 shows this impact.

As shown, this factor has a noticeable correlation with the run-time. Therefore, the

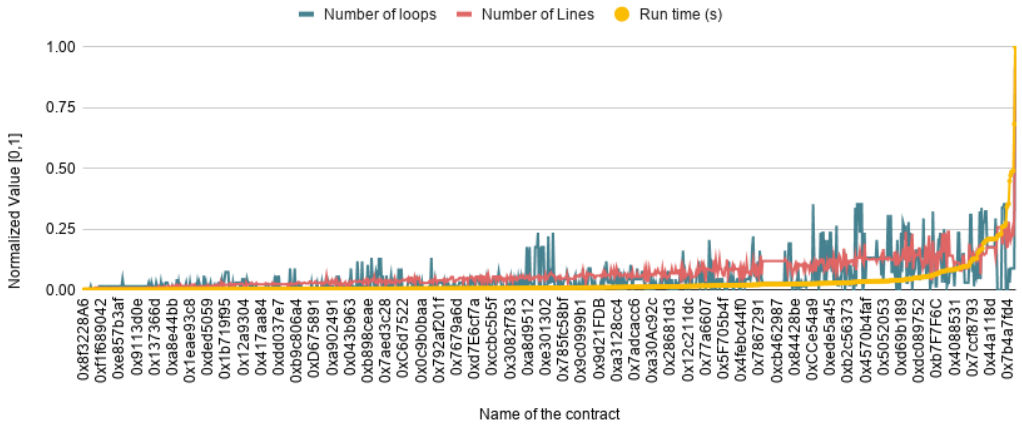

Figure 4.2: Effect of the Number of Loops and Line of Code on the Run-time

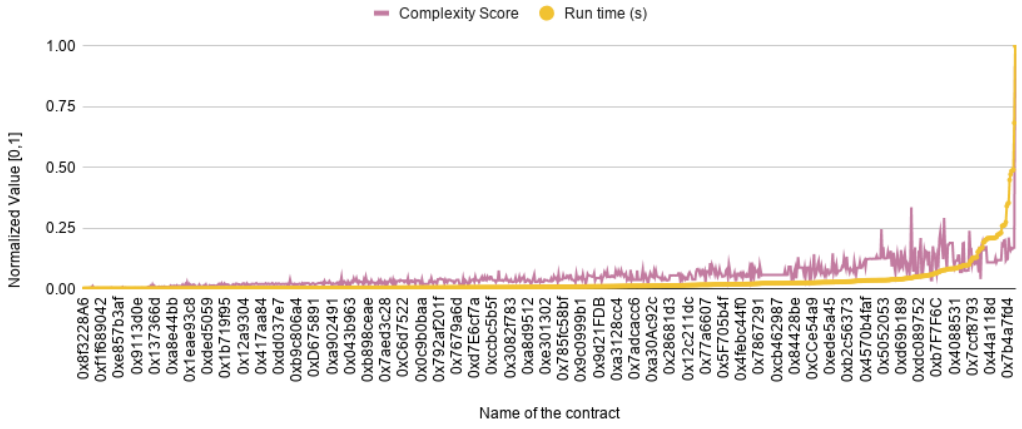

Figure 4.3: Effect of the Complexity Score on the Run-time

main factors in determining the run-time can be summarized into the "Complexity Score" of the code.

This experiment cannot be done on the other two parts of Gas Gauge. The reason is that the other parts contain run-time analysis. This means that the time to deploy and run the contracts has a significant impact on the run-time. Furthermore, for the Identification Phase factors like the search space, the ability of the fuzzer to find the right set of inputs, types of input variables, and the complexity of the target functions are essential factors that cannot be measured easily. For the Correction Phase the estimated value reported by the Threshold Estimator, complexity of the loop, and the actual threshold values are some of the factors that affect the run-time noticeably and cannot be measured easily. As a result, this experiment was only performed for the Detection Phase.

4.3 The Identification Phase

We performed two separate experiments for this part. The first experiment was done when designing the algorithm of the fuzzer, and the second was to evaluate the results of the Identification Phase.

4.3.1 Different Fuzzing Approaches

Three methods were considered when designing the whitebox fuzzer. The first approach was a random bit flip, as described before. The second one was a random byte flip, which flips every bit in the byte starting from the randomly chosen bit. Lastly, a random byte shuffle was tested. In this approach, the fuzzer chooses a random bit to flip, and then all the bits in the byte starting from the chosen bit get randomly shuffled. Figure 4.4 shows a chart comparing the three methods on a benchmark of 28 contracts containing a total of 31 functions with loops. We tested each method ten times on each function and recorded the average number of tries for each method until finding a set of inputs that causes the out-of-gas exception. The number of tries means the number of different combinations of inputs before finding a satisfying set. This number was bounded to fifty in our experiments in order to halt the process promptly.

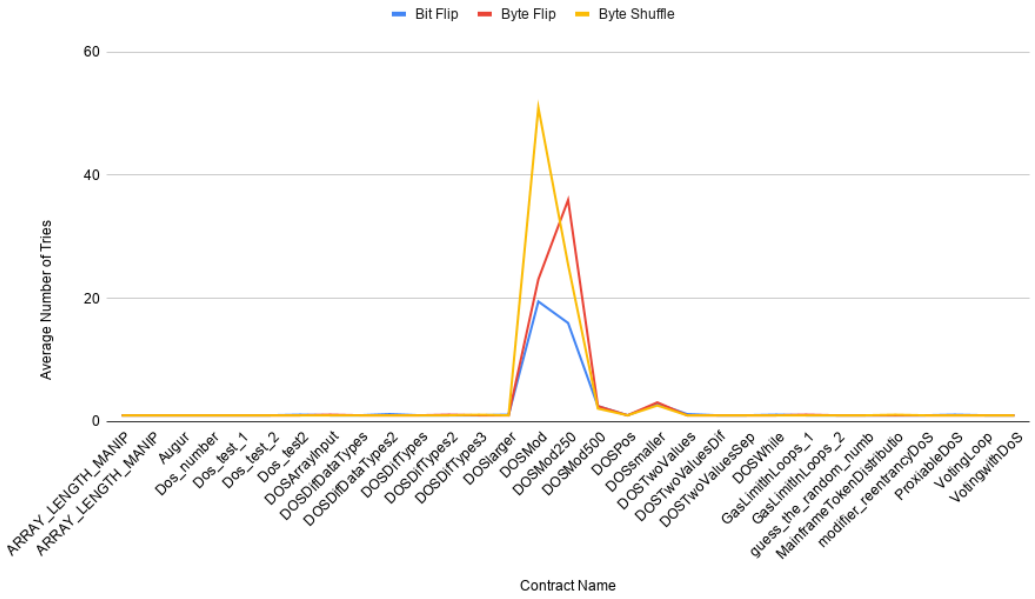

Figure 4.4: A Comparison of Different Methods for the Whitebox Fuzzer

During this experiment, we obtained the following results:

1. Because our implementation finds the exact functions and variables to fuzz, in most cases, all three methods can find the desired output within the first two tries. These are simple cases when one input variable is the loop bound, so the fuzzer has a high chance of finding the correct value for the value.

2. In some cases, when the bound is more complex, the fuzzer takes about 3 or 4 tries to find a correct set of inputs using any of the approaches. An example of this is when the difference between the values of two of the inputs is the loop bound.

3. There are also some cases that the fuzzer needs to try more numbers. An example of this is when "input mod 250" is the loop bound, and any number of loop iterations more than 240 triggers the out-of-gas condition. These contracts were the determining factors, and as shown, bit flip outperformed the other two methods. Hence, the bit flip approach was chosen in our design.

4.3.2 The Experimental Evaluation

In this experiment, we evaluated the results of the Identification Phase. Generally, finding an out-of-gas instance requires a run-time analysis based technique. To the best of our knowledge, Gas Gauge is the first tool, and the Identification Phase is the first algorithm that uses a run-time fuzzing technique to detect gas-related vulnerabilities and identify out-of-gas instances automatically. Most other fuzzers either do not identify gas-based vulnerabilities or require manual test cases. As a result, we did not find a direct competitor to compare the results of Identification Phase.

Therefore, we manually checked each contract containing loops to obtain the number of functions satisfying the condition of the fuzzer and the input variables affecting the loops. Only public functions containing loops and at least one input variable affecting the loop bounds satisfy the condition of the fuzzer. Only 979 functions in 501 contracts met the mentioned criteria. Then, we ran our tool on the benchmark and obtained the results. Lastly, we verified the results manually. The summary of the results is provided in Table 4.3.

Table 4.3: Summary of Identification Phase Results

	Number of Contracts	**Number of Functions**	**Average Run-time s)**
Total	1000	62750	-
Satisfying the Condition of the Fuzzer	501	979	-
Found by the Fuzzer	**499**	**968**	**53**
Out of Gas Instance Found by the Fuzzer	**331**	**614**	**61.5**

The fuzzer was able to detect 968 functions in 499 contracts and identify their variables correctly in 53 seconds per contract on average. Two of the contracts were not scanned by Slither, and as a result, the fuzzer was not able to identify 11 functions. The fuzzer was able to identify 614 instances of out-of-gas in 331 contracts. Although the fuzzer detected the rest, it could not find an out-of-gas instance in them for various reasons. Some of these reasons are that the fuzzer reached the maximum number of tries (was set to 10), the function contained structs or multi-dimensional arrays, the code reverted, or there was a required statement in the target function that was not satisfied. However, even when the fuzzer could not find an out of the gas instance, it still provided the function signatures with loops and variables affecting the loop bounds.

Overall, the fuzzer was able to detect all the correct functions in the contracts that Slither could scan and identified an out-of-gas instance for the majority of these functions.

Generally, fuzzers can only fuzz public functions. Furthermore, to find a set of input variables that cause the contract to run out of gas, at least one of the input variables should affect the loop bounds inside the function. We used a static analysis approach to cut down the action space accurately and efficiently. Thus, our fuzzer needs to only fuzz the public function containing loops affected by at least one of the input variables. As shown in Table 4.3, out of 62,750 available functions, only 979 met these criteria. Also, we statically detected the exact variables to fuzz. This caused the fuzzer to be able to find an out-of-gas instance within the first two tries in most cases as the search space was noticeably narrowed. If a general random fuzzer were used, the search space would be much larger, and as a consequence, the fuzzer would need many more tries. As a result, we did not compare our tool to a general random fuzzer.

4.4 The Correction Phase

In this experiment, we evaluated the results of the Correction Phase. Table 4.4 summarizes our benchmark, and Table 4.5 provides a summary of the results of Correction Phase.

Table 4.4: Benchmark Summary

Number of Contracts	1000
Total Number of Functions	62750
Number of Functions Containing Loops	3582
Number of Loops	4415

In the benchmark of 1,000 smart contracts, there are 4415 loops in 3582 functions. Gas Gauge was able to find the thresholds for 932 loops in 467 contracts. 779 of these thresholds were calculated using run-time verification with a high rate of accuracy, and 153 of these thresholds were estimated with an acceptable rate of accuracy. A threshold is estimated if it is greater than 5,000 or for any reason, the run-time verification is not able to find the correct number. The average run-time for each loop was about 389 seconds, which is much faster than manual processing.

To the best of our knowledge, Gas Gauge is the first tool ever attempted to find the loop upper bound limits in a smart contract. The closest tool to our work is GasTap/Gasol,

Table 4.5: Summary of Correction Phase Results

Number of Contracts Found	467
Total Number of Thresholds Found	932
Number of Thresholds Found by Run-time Verification	779
Number of Thresholds Found by the Estimator	153
Average Run-time Per Contract (s)	733
Average Run-time Per Loop (s)	389

which infers the upper bound limit of a function containing a loop. However, the format of their output is very different than ours. Also, based on our experiment in 4.2, the provided web interface does not support most of the contracts in our benchmark. Meanwhile, manually finding the thresholds is a tedious and time-consuming process.

Therefore, to get an idea of the accuracy of the calculated thresholds, we randomly picked 10 of the contracts and found the actual thresholds of their loops manually. Then, we obtained the values using Gas Gauge. Finally, we compared the manual results with the ones generated by Gas Gauge. Based on our results, the calculated threshold is normally about 2% lower than the actual threshold. We expect the calculated thresholds to be within 5% of the actual values and usually lower than the actual values since our modifications to the code introduce an insignificant extra gas usage.

Overall, the Correction Phase of Gas Gauge performed very well, which shows that it can be very handy to obtain loop thresholds in a smart contract.

4.5 Limitations of Gas Gauge

In many components of Gas Gauge Slither and Truffle Suite are used. Hence, Gas Gauge is limited by the limitations of these two tools. If slither is not able to scan a contract or

does not identify loops or data dependencies, all three algorithms of Gas Gauge miss some important information.

Also, the time limit and compilation problems of Truffle Suite may cause our tool to not operate properly. Functions that contain structs or multi-dimensional arrays are not supported by our tool as well. A struct is just a custom type that can be defined within a contract, and because it is different in each contract, it cannot be automated using Truffle Suite. Also, multi-dimensional arrays add an extra degree of complexity to the automation process, and because they are not common in Solidity contracts, they are not supported in our tool.

Finally, Gas Gauge requires access to the source code of the contract written in Solidity. Because we use run-time analysis in multiple sections of the tool, the contract has to be written in Solidity. Although Solidity is one of the most common programming languages used for smart contracts, there are other common languages that are not supported by our tool.

4.6 Related Work

There are multiple general smart contract verification tools available that scan a contract for multiple security vulnerabilities. The usual techniques in these tools are static analysis, symbolic execution, and fuzzing. Some of the most well-known tools that use symbolic execution, static analysis or a combination of both are Manticore [12], MPro [3], Mythril [8], Securify [9] (deprecated), Securify2 [10], Slither [6], SmartCheck [11], Verx [22], and Zeus [23]. These tools can detect multiple security vulnerabilities; however, most of them cannot identify gas-related vulnerabilities, or if they can, their result is not reliable due to their high false-negative rate. Fuzzer tools like ContractFuzzer [24], Echidna [25], and Harvey [26] either do not discover gas-related vulnerabilities or require manual test cases, which is a lengthy process.

Meanwhile, extensive research has focused on gas-related vulnerabilities of smart contracts. Gastap [17] derives gas upper bounds for all public functions of smart contracts via inferring size relations, generating gas equations, and solving the corresponding equations. Gasol [18] is an extension of Gastap [17]. It replaces the multiple accesses to the same storage location with one access. Also, GasCheker [27] mainly focuses on automatically identifying gas-inefficient code in smart contracts based on symbolic analysis to save the gas usage for the creation of a contract or the invoking process of the functions in a smart contract. GasFuzzer [28] proposes a gas-greedy strategy and a gas-leveling strategy to detect Exceptions Disorder kind of security vulnerabilities. Madmax [19] uses a

static program analysis technique to automatically detect gas-focused vulnerabilities. It first decompiles EVM bytecode using Vandal, then detects gas-related security problems from the decompiled code in Datalog language, which is a logic-based language. GasFuzz [29] adopts feedback-directed fuzz testing to automatically generate inputs, leading to a high gas consumption of contract functions. Additionally, eth-gas-reporter [30] reports gas usage per unit test in a solidity smart contract. It can be used to get the amount of gas used by calling a smart contract. However, most of these can not detect gas-related DoS attacks directly, or they do not provide any information to fix the problem.

Gas Gauge can find all the loops in a contract reliably and quickly. Also, it identifies the exact functions and variables and provides an out-of-gas instance that helps developers investigate the problem further. Finally, to the best of our knowledge, no other tool provides a contract's loop thresholds. Gas Gauge is the only tool that accurately and reliably finds the threshold values and provides more useful information like each loop's type, the variables affecting it. This information is extremely helpful in order to repair the code and prevent gas-based attacks like DoS with Block Gas Limit.

Chapter 5

Conclusions and Future Work

Conclusions:

Smart contracts are an essential application of blockchain. Because they involve monetary transactions, it is crucial to make sure the contracts are risk-free. Many tools scan and identify different vulnerabilities of smart contracts. One of the well-known vulnerabilities the state-of-the-art tools cannot identify reliably is DoS with Block Gas Limit.

This thesis summarizes the design and implementation of Gas Gauge, an automatic tool that helps developers and contract owners identify DoS with Block Gas vulnerability and repair their code. This tool contains three powerful algorithms. The first one is a static analysis approach that summarizes a contract and detects all the contract loops. The second method is a whitebox fuzzeing based on static analysis and run-time analysis. It identifies the public functions vulnerable to DoS With Block Gas limit. Since it uses run-time fuzzing, it has zero false positives. Also, it uses an advanced static and run-time analysis so, it has a low false-negative and is very efficient. The last algorithm is a static analysis and run-time verification method that determines the point at which the contract starts to go out of gas. Because it uses run-time verification, the results are incredibly accurate. The method is also optimized using a binary search approach and an independent parallel processing design to make it faster and more efficient.

Finally, our experimental evaluation results on 2,000 real-world Solidity smart contracts show that all the three algorithms of Gas Gauge are extremely accurate and efficient. This implementation and the results demonstrate that Gas Gauge is reliable and incredibly useful.

Future Work:

The current implementation only supports contracts written in Solidity. This is because the code uses the source-code to extract useful information and utilizes the Truffle Suite to performs run-time verification. As an improvement, we can modify our tool so that it can support more programming languages like Vyper and JavaScript.

Also, the fuzzer can be improved so it can identify the private functions containing loops and fuzz the public functions that call those private ones.

Furthermore, the Correction Phase uses a binary search to cut down the search space quickly. However, it can be improved by adding a feedback-based approach to it. We can use the amount of the remaining gas after each execution to guide the binary search to cut down the search space even further.

Bibliography

[1] ConsenSys, "Known Attacks," Known Attacks - Ethereum Smart Contract Best Practices. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks [Accessed: 19-Apr-2020]

[2] Etherscan. https://etherscan.io, [Accessed: 11-Nov-2020].

[3] W. Zhang, S. Banescu, L. Pasos, S. Stewart, and V. Ganesh, "MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 2019.

[4] [Online]. Available: https://swcregistry.io/ [Accessed: 19-Apr-2020].

[5] Godefroid, P., Levin, M. Y., Molnar, D. (2012, January 11). SAGE: Whitebox Fuzzing for Security Testing. Retrieved November 19, 2020, from https://queue.acm.org/detail.cfm?id=2094081

[6] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019.

[7] Truffle Suite, "Sweet Tools for Smart Contracts," Truffle Suite. [Online]. Available: https://www.trufflesuite.com/. [Accessed: 29-May-2020].

[8] ConsenSys, Mythril-classic," https://github.com/ConsenSys/mythril., accessed: 15-March-2019.

[9] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bunzli, and M. Vechev, Securify: Practical security analysis of smart contracts" in Proceedings of the ACM Conference on Computer and Communications Security. Association for Computing Machinery, oct 2018, pp. 67 - 82.

[10] B. Chess and G. McGraw, Static analysis for security," IEEE security privacy, vol. 2, no. 6, pp. 76-79, 2004.ChainSecurity, Securify 2.0," https://github.com/eth-sri/securify2., accessed: 15-March-2019.

[11] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, Smartcheck: Static analysis of ethereum smart contracts," in Proceedings of the 1st International Work- shop on Emerging Trends in Software Engineering for Blockchain, 2018, pp. 9 - 16.

[12] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engi- neering, ASE 2019. Institute of Electrical and Electronics Engineers Inc., nov 2019, pp. 1186 - 1189.

[13] "Quantstamp Audit Launcher," Quantstamp Audit Launcher. [Online]. Available: https://certificate.quantstamp.com/. [Accessed: 29-May-2020].

[14] Grincalaitis, M. (2017, November 19). The ultimate guide to audit a Smart Contract + Most dangerous attacks in Solidity. Retrieved November 18, 2020, from https://medium.com/ethereum-developers/how-to-audit-a-smart-contract-most-dangerous-attacks-in-solidity-ae402a7e7868

[15] Solidity. (n.d.). Retrieved November 19, 2020, from https://docs.soliditylang.org/en/develop/contracts.html

[16] "Units and Globally Available Variables," Units and Globally Available Variables - Solidity 0.5.3 documentation. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/units-and-global-variables.html. [Accessed: 19-Apr-2020].

[17] E. Albert, P. Gordillo, A. Rubio, et al. GASTAP: A gas analyzer for smart contracts[J]. CoRR, vol. abs/1811.10403, 2018.

[18] E. Albert, J. Correas, P. Gordillo, et al. Gasol: Gas analysis and optimization for ethereum smart contracts[C]//International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Cham, 2020: 118-125.

[19] N. Grech, M. Kong, A. Jurisevic, et al. Madmax: Surviving out-of-gas conditions in ethereum smart contracts[J]. Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA): 1-27.

[20] Contract Library. https://contract-library.com, [Accessed: 10-Feb-2021].

[21] ConsenSys, "ConsenSys/solidity-metrics," GitHub. [Online]. Available: https://github.com/ConsenSys/solidity-metrics. [Accessed: 13-Feb-2021].

[22] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "VerX: Safety Verification of Smart Contracts," 2020 IEEE Symposium on Security and Privacy (SP), 2020.

[23] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts," Proceedings 2018 Network and Distributed System Security Symposium, 2018.

[24] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: fuzzing smart contracts for vulnerability detection," Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018.

[25] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020.

[26] V. Wüstholz and M. Christakis, "Harvey: a greybox fuzzer for smart contracts," Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020.

[27] T. Chen, Y. Feng, Z. Li, et al. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts[J]. IEEE Transactions on Emerging Topics in Computing, 2020.

[28] I. Ashraf, X.Ma, B. Jiang, et al. GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities[J]. IEEE Access, 2020.

[29] F. Ma, Y. Fu, M. Ren, et al. GasFuzz: Generating High Gas Consumption Inputs to Avoid Out-of-Gas Vulnerability[J]. arXiv preprint arXiv:1910.02945, 2019.

[30] eth-gas-reporter. https://github.com/cgewecke/eth-gas-reporter, [Accessed: 10-Feb-2021].