

A Scalable Method to Analyze Gas Costs, Loops and Related Security Vulnerabilities on the Ethereum Virtual Machine

MICHAEL KONG

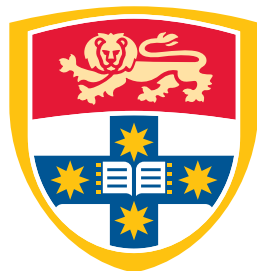
SID: 312142668

Supervisor: Dr. Bernhard Scholz

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Information Technology (Honours)

School of Information Technologies
The University of Sydney
Australia

7 November 2017



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

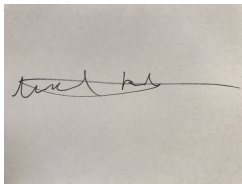
I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name:

Michael Kong

Signature:

A rectangular photograph of a handwritten signature in black ink on a light-colored surface. The signature is cursive and appears to read 'Michael Kong'.

Date: 07 November 2017

Abstract

Ethereum is a distributed blockchain platform that executes programs stored on the blockchain known as smart contracts. These smart contracts tend to have limited functionality, although they can be arbitrarily complex. The Ethereum Virtual Machine (EVM) is a simple stack machine that holds up to 1024 elements at any time. This machine executes transactions across each node on the network which are validated by the miners via a consensus algorithm known as "Proof of Work". Because transactions on the blockchain are considered "immutable", it is important that programs behave in the way they are intended to behave. Vulnerabilities may arise from incorrectly written contracts, allowing users to interact with a contract in a way not intended by the developer. These vulnerabilities can lead to devastating exploits, such as the hack of the TheDAO and the Parity multi-signature smart contract, which lead to the loss of over USD\$50 million and USD\$30 million of Ether respectively. Measuring gas costs, the costs of performing computations on the network, is needed to both improve the efficiency of smart contracts and detect resource-based vulnerabilities that can consume all the gas in a transaction. Performing this analysis is challenging, as it requires a deep technical understanding of how both the EVM and smart contract functionality works. In this paper, we create an accurate and scalable method to analyze gas costs. We also use an integer linear program and an algorithm using shortest-paths to calculate the "maximum frequency" of an edge in our control-flow graph, allowing us to determine the upper bound in the number of iterations of a loop for a given gas budget. In addition, we build logic specifications in datalog using the Soufflé engine using maximum frequencies to detect common gas-related vulnerabilities in smart contracts. Buterin (2016) stated: "Formal verification can be layered on top. One simple use case is as a way of proving termination, greatly mitigating gas-related issues" (3). This thesis attempts to address these issues.

Acknowledgements

I would like to thank my supervisor, Dr. Bernhard Scholz for his tireless effort, patience and help throughout the year. Our in-depth discussions gave me many ideas that were crucial in designing solutions to solve several open problems in smart contract security. I would also like to thank Anton Jurisevic, Lexi Brent and Eric Liu for their contributions to the Vandal framework which this work builds upon, and their advice and suggestions to my thesis throughout the year.

CONTENTS

| | |
|---|------------|
| Student Plagiarism: Compliance Statement | ii |
| Abstract | iii |
| Acknowledgements | iv |
| List of Figures | vii |
| List of Tables | ix |
| Chapter 1 Introduction | 1 |
| 1.1 Problem Statement | 4 |
| Chapter 2 Resource Exploits | 7 |
| 2.1 Message Calls | 9 |
| 2.2 GAS costs | 10 |
| 2.3 Exploits | 14 |
| 2.4 Dynamic Loop Sizes | 14 |
| 2.5 Wallet Griefing | 16 |
| 2.6 Mass Clearing of Storage | 18 |
| 2.7 Summary | 19 |
| Chapter 3 Analysis | 21 |
| 3.1 General framework | 23 |
| 3.2 Loops | 24 |
| 3.3 Wallet Griefing | 30 |
| 3.4 Mass Clearing of Storage | 32 |
| 3.5 Maximum Frequency Analysis | 34 |
| 3.6 Transforming the CFG | 35 |
| 3.7 Iterative Solution | 36 |
| 3.8 Integer Linear Solution | 39 |

| | |
|--|-----------|
| 3.9 Logic of Resource Exploits | 40 |
| Chapter 4 Experiments | 43 |
| Chapter 5 Results | 46 |
| Chapter 6 Future Work and Discussion | 51 |
| Chapter 7 Related Work | 53 |
| Chapter 8 Conclusion | 56 |
| Bibliography | 58 |
| Appendix A Four Contracts Analyzed | 62 |
| Appendix B Commands to run analysis | 63 |
| Appendix C Screenshot of Python maximum frequency scripts | 66 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | How new smart contracts are deployed on the blockchain | 2 |
| 2.1 | A simple message call between a malicious and victim contract. Note that the contract being called can execute arbitrary code in its fallback function, which could include throwing, causing the rest of the execution to fail, or perform recursive calls to exploit the calling contract's functionality. | 8 |
| 2.2 | A simple message call example using the transfer(x) function, where x is some amount of Wei | 10 |
| 3.1 | Security Analysis Pipeline | 24 |
| 3.2 | The control flow graph of a simple function with a loop | 25 |
| 3.3 | The control flow graph of a simple function with a loop bound determined by external input. | 28 |
| 3.4 | The control flow graph of a simple function with a message call inside the loop | 31 |
| 3.5 | <i>C</i> allowing a user to clear an array of addresses | 33 |
| 3.6 | Datalog Program: Specification for dynamic bound. | 41 |
| 5.1 | The number of times a vulnerability has been detected in a smart contract | 46 |
| 5.2 | The number of contracts with a given number of vulnerabilities | 47 |
| 5.3 | Cumulative Soufflé analysis time across all analyzable smart contracts | 48 |
| 5.4 | Performance of ILP vs. iterative solution across a random sample of 25000 contracts. The blue line (bottom) is the performance of the ILP, the green line (top) is the performance of the iterative solution. | 50 |
| 5.5 | (Left to right - owned, ownable, tokenholder and Token contracts) ILP vs. Iterative solutions, with time (in seconds) of the y-axis and the number of edges on the x-axis, sorted by time per edge. The blue line (top) is the performance of the ILP, the green line (bottom) is the performance for the iterative solution. | 50 |

| | | |
|-----|--|----|
| B.1 | Sample TAC output using the token contract | 64 |
| B.2 | Sample maximum frequency output using the token contract | 65 |
| C.1 | Screenshot of code that creates a common start and end node, and converts basic blocks into weighted edges | 66 |
| C.2 | Screenshot of the function that returns the maximum frequencies per node using Dijkstras' algorithm | 67 |

List of Tables

| | | |
|-----|--|----|
| 5.1 | Summary of Soufflé and Decompilation Analysis | 47 |
| 5.2 | Summary of Maximum Frequency Analysis | 49 |
| A.1 | Summary of owned.hex ILP vs. Iterative Performance | 62 |
| A.2 | Summary of ownable.hex ILP vs. Iterative Performance | 62 |
| A.3 | Summary of TokenHolder.hex ILP vs. Iterative Performance | 62 |
| A.4 | Summary of Token.hex ILP vs. Iterative Performance | 62 |

CHAPTER 1

Introduction

Ethereum is a blockchain inspired by Bitcoin, the first cryptocurrency developed and released by an individual (or individuals) under the pseudonym “Satoshi Nakamoto” in 2009. The main purpose of Bitcoin was to create “a purely peer-to-peer version of electronic cash” (30). In contrast, Ethereum’s aim is arguably larger - to create a distributed computer that can execute arbitrarily complex contracts. These are often referred to as “smart contracts”, and can be used to represent real-life entities and agreements, such as organizations, crowd funding campaigns, properties and deeds, among many others including allowing people to create their own versions of electronic cash governed by the rules of the contract (often known as “programmable money”).

Smart contracts are programs that are stored on the blockchain. They have a unique address, permanent storage and persistent memory. Accounts on Ethereum can communicate to a smart contract by using its public address and specifying what function to call. To create a smart contract, a developer writes a program using a high-level language that compiles to bytecode, which represents a hash of all the contract’s properties. This hash is then sent in a transaction by an address to the blockchain, which miners include in a block.

All operations executed by smart contracts and transactions require gas, a concept that Ethereum invented. The purpose of gas is to ensure that users of the network pay for their computations, creating the incentive to run transactions as efficiently as possible. The address initiating a transaction gives a gas budget, a finite quantity representing the total amount of gas the address is willing to spend in order to complete the transaction. Hence, computations are resource limited - they can only be performed up to the gas budget.

To pay for gas, an address needs to forward some Ether in the transaction. Ether (ETH) is a cryptocurrency used to pay for computations done on the network, and can be traded on multiple exchanges and between different users in exchange for fiat currency or other cryptocurrencies (20). Ether is converted

to gas according to the conversion rate specified by the `gasPrice`, which converts one unit of gas to a given number of `Wei` (one `Wei` equals one over eighteen-zeros worth of `ETH`). Miners are incentivized to include transactions in the blockchain with a higher `gasPrice`, as they earn more `ETH`.

The amount of `Wei` forwarded with a transaction is given by the formula (8):

$$Wei = gasPrice * gas_budget \quad (1.1)$$

Ethereum is considered to be a Turing complete machine, as smart contracts can execute any kind of computation. Hence, the Turing completeness of Ethereum increases the complexity and potential for vulnerabilities to exist in smart contracts. Due to theoretical limitations (e.g. the halting problem), there are no means to fully automatically check the correctness of smart contracts. Practically, however, static analysis is applicable (with the acceptance of false positives) to find vulnerabilities. This is achieved by using graph theory to map patterns that may be indicative of a vulnerability at the EVM-level to logical statements. Thus, no matter the complexity of the code and what is written in a higher-level language, static analysis should be able to detect vulnerabilities that may appear subtle to the developer.

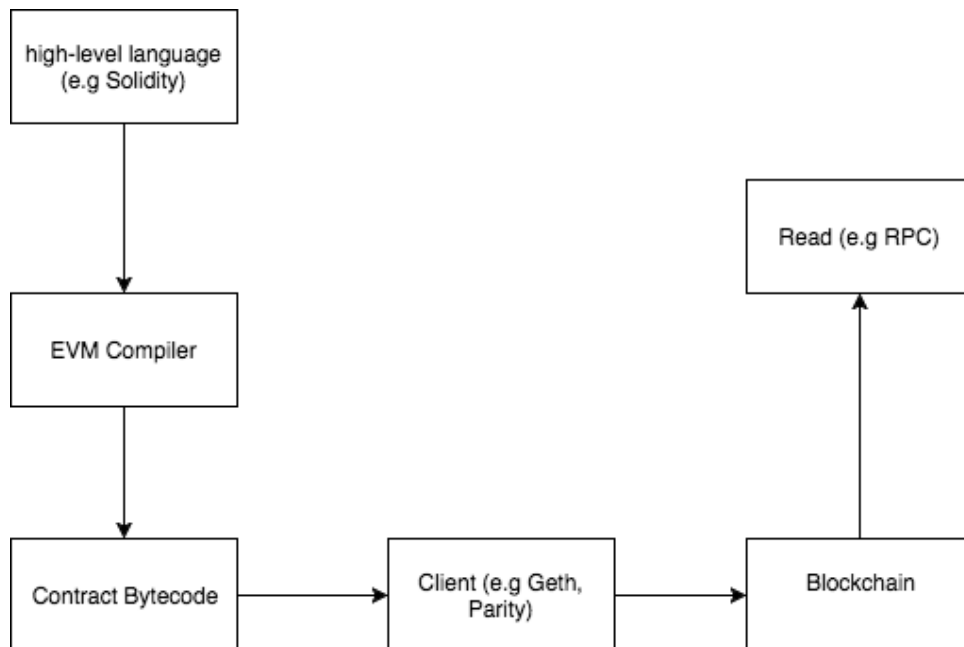


FIGURE 1.1: How new smart contracts are deployed on the blockchain

A typical work-flow for deploying smart-contracts written in Solidity (19) is shown in Figure 1.1.

A developer writes a program in a higher level language such as Solidity. This program is run through a compiler that translates the higher level code into Ethereum bytecode (a string of characters). This contract is sent to the network via a client such as Parity or Go-Ethereum. The miners then include the contract bytecode into a particular block that is mined and validated, becoming part of the blockchain. The contract now exists publicly for individuals to interact with using services such as `web3js`, which performs remote procedure calls (RPC) to the blockchain via a client.

When a transaction is created, it consists of the following mandatory parameters:

- (1) **From:** Address the transaction is being sent from.
- (2) **To:** Address the transaction is being sent to.
- (3) **gas:** The amount of gas to be forwarded in the transaction (a positive integer). Note that unused gas is refunded, but all gas is consumed if an exception is thrown. This is the **gas budget**.
- (4) **gasPrice:** The conversion rate between gas and `Wei`. All transactions are paid for in ETH, where one ETH equals one followed by 18 zeros `Wei`. The higher the `gasPrice`, the more likely a miner will include your transaction in their block. Note that `gasPrice` is completely separate from the gas cost in the EVM level.

There is a gap between high-level source code in Solidity and actual resource costs of the translated solidity program running on the EVM. It is challenging to determine the output of a given transaction without a detailed understanding of the underlying EVM, as most developers write their code in a higher level language such as Solidity without sufficiently understanding the technology(2). Yet gas costs are determined purely by the EVM, and not high-level languages that compile to it.

Poorly-written contracts may lead to certain execution paths that consume a large quantity (if not all) of the gas sent in a transaction, and may expire prematurely. The premature termination of smart contracts, hence, may lead to exploitable program states of smart contracts. For each instruction (i.e `OP CODE`) executed, the gas budget is depleted by a certain amount depending on the instruction executed. When the gas budget is spent, an “out-of-gas” exception is automatically thrown. An exception causes the EVM to execute a `JUMP` instruction to an invalid destination on the stack. As a consequence, all changes to the state by the transaction are rolled back. Note that an exception may occur for other reasons, including if the wrong parameters are specified when calling a function or if the instruction `THROW` is present. In all cases, the entire gas budget is consumed and paid to the miner of the block. Measuring

gas is important, as a user would like to send as little gas as possible as there is always a risk that all of the gas might be consumed in an exception.

Many exploits occur due to contract-to-contract communication which can lead to issues such as race conditions, as a smart contract developer does not know what code is contained in other contracts that may interact with theirs. In terms of gas, an external contract may consume an arbitrary amount of gas when called because the program's execution is given to the external contract, who may execute whatever code is written in their smart contract. Thus, a developer must assume that any contract-to-contract communication could, in the worse case, lead to an exception being thrown. Failure to consider this may lead to vulnerabilities as described below.

Furthermore, gas constraints limit the size of loops. A loop may become too big that iterating through it requires a gas budget bigger than the block gas limit (the limit set by the network itself). Thus, we want to determine if a loop bound is too large, or if a pattern exists such that a loop bound *may* become too big overtime, for example, if the bound is determined by the number of addresses stored as an array in the smart contract, where an address can add itself to that list by calling another function. For example, a contract called `Government` had 1100 ETH (worth USD\$340,000) stuck in it as in order for the winner to claim the jackpot, an array full of participants needed to be cleared, where, totaling 5.5 million, the operation consumed more gas than the block gas limit itself! (15) (14). In addition, McCorry, Shahandashti and Feng Hao (2017) demonstrated that their privacy-preserving voting system allowed only 60 votes to be processed in a loop before hitting the then block gas limit of 4.7million, requiring votes to be batched across multiple transactions(27).

1.1 Problem Statement

This work describes a more accurate method of statically calculating gas costs with each statement, and hence the overall cost of each execution path within a smart contract. Furthermore, we introduce a potential new class of exploits stemming from the resource constraints of smart contracts. As previously mentioned, these new exploits can be a significant threat for smart contracts stored on the block chain, effectively freezing the smart contract's functionality and the tokens stored within it. These threats come from external contract calls and the assumptions made by programmers that their smart contract can deal with ever-growing storage without realizing that loops or the reassignment of the storage values are expensive.

Resource-based exploits include the following three:

- (1) **Unbounded and Dynamically-bounded Loops:** If a loop is unbounded or whose bounds dynamically grows as a result of the growth in storage or memory, then it is guaranteed to reach a point where the cost of executing the loop will exceed the block gas limit.
- (2) **Wallet Griefing:** If a call to an external contract occurs within a loop, that loop might throw an exception and cause the entire transaction to roll back, meaning the loop may never complete.
- (3) **Mass reassignment of Storage:** If storage is initially set or re-set and the current storage is too large, then such an operation might exceed the block gas limit.

The contribution of this work is as follows:

- (1) A new class of exploits that rely on the consumption of gas for each executed statement where the gas budget of a program is always finite and positive.
- (2) A new maximum frequency analysis for statements in smart contracts. The maximum frequency analysis is expressed as a mathematical program. We first executed it as an Integer Linear Program and later found that this problem can be solved using an algorithm that runs in polynomial time based on shortest path searches using Dijkstra's algorithm.
- (3) A new semantic description of resource exploits expressed in logic that captures potential resource exploits using data-flow, control-flow, and maximum frequency analysis results. The logic of the resource exploits is executable by a logic synthesis tool called Soufflé. The synthesis tool produces a highly efficient static program analysis tool in C++.
- (4) Implementation of the proposed static analysis to mathematically determine if a given resource-vulnerability exists within a particular smart contract.
- (5) Experimental data demonstrating the performance of our polynomial algorithm to the Integer Linear Program on most smart contracts currently deployed on the Ethereum Mainnet blockchain, the number of contracts with loops in them, the accuracy of our logical specifications and the number of existing smart contracts that have resource-exploits present.

The organization of this thesis is as follows. In chapter 2 we introduce a new class of exploits that take advantage of the finite resource allotted to a smart contract. In chapter 3 we introduce new means to find resource exploits and the concept of maximum frequencies and demonstrate two methods of calculating them, modeling it first as an integer linear problem, and then deriving a polynomial algorithm. We also describe how this can be used in our exploit analysis. In Chapter 4 we provide experimental data for

the Ethereum blockchain. In Chapter 6 we outline some suggestions to improve our framework, and in Chapter 7 we survey related work.

Resource Exploits

A vulnerability is a weakness in a smart contract that, when taken advantage of, may lead to an unexpected or undesirable result. Exploits use vulnerabilities to cause some damage to the participants of the smart contract, most commonly denying access to the smart contract or stealing tokens such as ETH stored there.

Most exploits occur due to the ability of smart contracts to interact with one another via message calls. A message call is an object that allows data and the execution of a transaction to pass from one smart contract to another temporarily. The execution path returns to the original after a value is returned. A certain amount of gas is also forwarded to that contract address, which is subtracted from the original gas budget and the remaining forwarded budget is refunded to the caller when the message call is returned. This allows another smart contract to perform some execution and then return to the original contract, or even call another contract (leading to a chain of message calls between multiple smart contracts). The main purpose of message calling is to increase the functionality and flexibility of smart contracts by allowing smart contracts to communicate with each other and pass data back and forth.

However, because a message call allows the target contract address to execute *arbitrary* code (within the confines of the forwarded gas budget), the contract may have some malicious code that causes unexpected behaviour. The *malicious contract* may manipulate the state of the *victim contract* causing the transaction to fail unexpectedly (as in the case of Wallet Griefing described below) and may even prevent all transactions incorporating that execution path from completing.

As every instruction in a transaction needs to be executed, the called contract is forwarded an amount of gas g_f deducted from the gas budget b . The `OP_CODE_gas` returns the remaining b at that point in the transaction's execution, which can be forwarded onto the called contract:

$$g_f = b - c \tag{2.1}$$

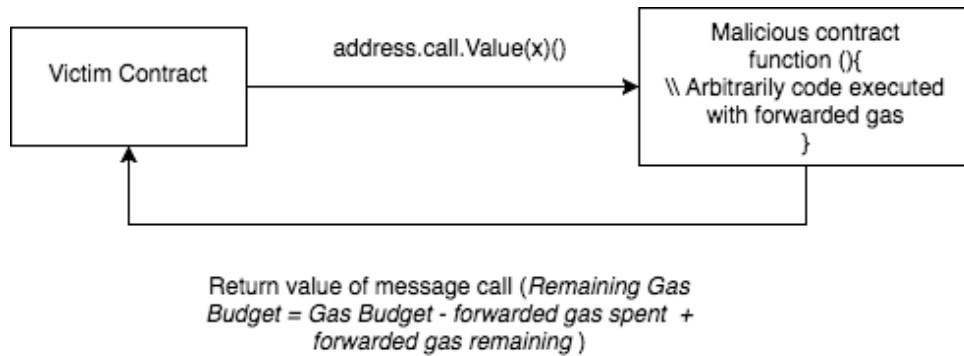


FIGURE 2.1: A simple message call between a malicious and victim contract. Note that the contract being called can execute arbitrary code in its fallback function, which could include throwing, causing the rest of the execution to fail, or perform recursive calls to exploit the calling contract’s functionality.

where c is the gas consumed by the external contract.

However, by default, if a message call is generated by the Solidity function `send(x)` or `transfer(x)`, where x is some amount in Wei , then the forwarded gas amount is the “gas stipend”, fixed at 2300 Gas. Note that this is also deducted from $b(19)$. If the gas stipend plus the 21000 gas required for a transfer of Wei is greater than b , then the execution will throw.

Note that if all of g_f is consumed, then the execution will throw as well, no matter the remaining b . This means, for example, that any contract you call could always throw, thus revering state changes to the blockchain. Given that you are likely not to know what contracts will call your contract (unless you have explicitly “white-listed” their addresses, meaning only those contracts can call certain functions in your contract), then a developer needs to assume that any message calls *could* lead to an exception being thrown.

Inefficient coding patterns can also exist with the smart contract’s internal execution paths. The execution of a function within a smart contract does not necessarily consumes a fixed amount of gas. The gas consumption of a contract depends on the data and control flow of smart contracts. For example, a smart contract may also allow loops to be bounded by arrays that grow in size over time as more entries are added, for example if a user adds an address to an array which keeps track of all members of a particular organization represented by the contract. A smart contract may also attempt to clear this array by changing all values to zero, which requires a significant amount of gas (as described below). Although

these problems are not caused by an attack or malicious contract, they are weak coding patterns which quickly lead to the consumption of all gas.

As a consequence the analysis of the gas cost per contract is hard because we not only need to count the number of operations that take place, but also consider loops and other execution paths (such as recursion) that cause certain operations to *repeat*, thus consuming gas. This thesis attempts to identify these coding patterns and notify the user of vulnerabilities that may arise from them.

2.1 Message Calls

A message call is the event of calling a smart contract from another contract. When a contract receives a message, it will decode any data sent and the EVM will perform its operations in the context of that contract and the message will be "acted upon"(51). A transaction is essentially a write-operation to the blockchain consisting of a collection of message calls that take place. Note that the miners do not care about the number of message calls within a transaction, as only transactions can lead to permanent changes in the state on the blockchain. When a transaction is sent to the Ethereum network via a node, all nodes on the network check to make sure the transaction is valid (i.e the execution of EVM operations for a set of instructions does not lead to an exception). The transaction is then mined by a miner into a block. The block is then pushed into the blockchain, along with a records of any changes to state (for example, changing the value of some storage).

When a message call is created and sent, the input data consists of three parts(36):

- (1) **4 bytes in length** - A Keccak hash of the function signature (the function of the smart contract the transaction is targeting).
- (2) **32 bytes in length** - The 20 byte address the transaction is sending to (note that 12 extra 0s added to the left of the address to create 32 bytes).
- (3) **32 bytes in length** - The `msg.data` to send across.

There are two types of addresses in Ethereum: a wallet address and a contract address. A wallet address can be used to store Ethereum and other tokens run on top of the Ethereum network and interact with other wallet or contract addresses. A contract address represents a smart contract that other wallet or contracts can communicate with and execute the code written in the smart contract itself. Each address is exactly 160-bits long, and is used to identify accounts on the blockchain.

When a contract performs a message call to a wallet, by-default the function signature is blank. Because wallets have no fallback function, no extra computation is performed by the wallet itself.

However, when a contract performs a message call to another wallet, if a function signature matches the signature of a function in the external contract, the function's computation is executed. If the function signature is blank, the contract will execute the contract's fallback function (if it exists) which may include any amount of arbitrary computation. It is this *arbitrary computation* that leads to a large number of potential vulnerabilities. A vulnerable contract, depending on the desired execution, may be vulnerable to a malicious contract that execute some code that takes exploits the vulnerability. An example is theDAO hack, where a malicious contract was able to keep executing the same `withdrawDAO()` function by placing the function in the malicious contract's fallback function, so that it would be called repeated after the `call.value()` message call was executed. This exploit is known as "reentrancy", and will not be explored in this paper but has been previously implemented in the logical specifications of the original security analysis pipeline as described in Figure 1.1.

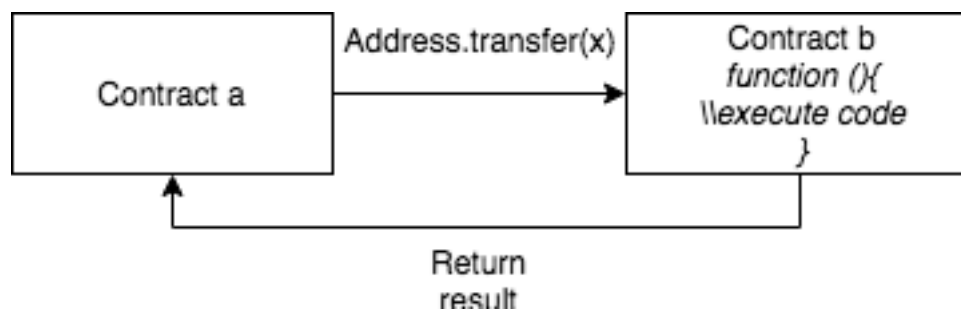


FIGURE 2.2: A simple message call example using the `transfer(x)` function, where `x` is some amount of Wei

2.2 GAS costs

Gas is a “unit of payment” used by the address initiating a transaction to pay for the execution of OP CODES by other nodes on the network and reward the miners for confirming the result. The costs of each OP CODE are described in the “Ethereum Yellow Paper”, a technical specification kept up to date with the latest version of the network(51). Some operations have a fixed gas costs. For example, the ADD OP CODE has a gas cost of 3. However, some operations have a variable gas cost according to a specific formula and the data it is dealing with. Some of these variable costs and how our static analyzer deals with them are described below. There is also a gas-limit per block, which represents the maximum

amount of gas that can be spent in a single block. Code written in a higher-level languages such as Solidity and compiled to bytecode. Essentially, the code provides instructions to the EVM, instructing it to execute a series of OP CODES. Gas is deducted from the gas budget for each OP CODE executed by the EVM in the set of instructions.

A user cannot specify any arbitrary amount for the gas budget b , as the Ethereum network imposes specific constraints on how much gas can be spent on a single transaction. These represent both a lower and an upper-bound to the amount of computation that is performed by the network.

All transactions, no matter what computation is performed, will consume a minimum of 21000 gas plus the cost of executing any function specified in the transaction's function signature. This cost arises because it is the amount of gas spent in order to send a transaction from one address to another(51):

$$G_{transaction} = 21000 \quad (2.2)$$

Because 21000 is paid initially for any transaction made, and executing state changes to the blockchain requires a transaction, our gas analyzer will automatically *deduct* 21000 gas when analyzing a smart contract. Note that neither Chen *et al* (2017) or Luu *et al* (2016) take this into account when performing their own analysis of gas consumption. A user may be misled to believe that a certain execution path in a transaction took some amount of gas, without realizing that 21000 gas was already consumed simply by submitting a transaction, regardless of the execution path it traverses.

Hence, if a user attempts to set $b < 21000$, the gas analyzer will print an error to the console:

```
ERROR: Minimum 21000 gas needs to be specified for the gas budget.
```

There is also a gas limit per block agreed upon by the miners known as the *block gas limit*. The block gas limit is the amount of computation that can be included in any one block. Hence, if a user attempts to perform a transaction that exceeds the block gas limit, an exception will be thrown:

```
Error: Exceeds block gas limit
```

A transaction needs to conform to the following inequality as a necessary condition for the transaction to be valid (51):

$$T_g \leq B_{Hl} - \ell(B_{\mathbf{R}})_u \quad (2.3)$$

Where:

- (1) T_g : The `gasLimit` for a given transaction T specified by the caller.
- (2) B_{Hl} : The current block gas limit.
- (3) $\ell(B_{\mathbf{R}})_u$: The gas that has been consumed by prior transactions for the given block B .

Note that $\ell(B_{\mathbf{R}})_u$ cannot be known in advance using static analysis. Hence, we assume that no prior transactions have been entered into the block when determining the block gas limit (i.e we assume $\ell(B_{\mathbf{R}})_u = 0$). A transaction which is less than or equal to the block gas limit can still be included into a block, assuming that the miner accepts that transaction, and only that transaction. Hence, we check that:

$$T_g \leq B_{Hl} \quad (2.4)$$

The block gas limit is also dynamic and changes after each block is mined depending on how the miners vote, as they ultimately determine the gas limit. However, the protocol only allows miners to change the block gas limit within certain bounds. These changes follow the semantics as outlined by Wood (2017) (51). Hence, the gas limit H_l of the block H must fulfill the following relations:

$$H_l < P(H)_{Hl} + \left\lfloor \frac{P(H)_{Hl}}{1024} \right\rfloor \quad \wedge \quad (2.5)$$

$$H_l > P(H)_{Hl} - \left\lfloor \frac{P(H)_{Hl}}{1024} \right\rfloor \quad \wedge \quad (2.6)$$

$$H_l \geq 125000 \quad (2.7)$$

where $P(H)_{Hl}$ is the previous block.

Because we are interested in finding the upper gas limit, we assume the *worst case* when it comes to determining the block gas limit, and hence we consider the equation:

$$H_l = P(H)_{Hl} + 1 - \left\lfloor \frac{P(H)_{Hl}}{1024} \right\rfloor \quad (2.8)$$

Note that this assumes that **all** of the miners will vote to decrease the block gas limit by the maximum possible amount, and thus represents the *minimum* block gas limit possible for the next block. Hence, we state that H_l is the current block gas limit we will test against. Checking against the block gas limit

is important given that the gas limit has two restrictions. Firstly, it represents the **hard** upper gas limit any transaction can take, regardless of the execution paths in a given smart contract or the amount of gas forwarded in the transaction. Otherwise, the transaction will never be mined by the network. Secondly, it can change significantly in a short period of time, thus changing the amount of computation allowed by any single transaction. For example, between 9:20 UTC and 11:00 UTC on 29 June 2017, miners voted to increase the gas limit by 33% from roughly 4.7 million to 6.3 million gas units per block(25).

Finding the current block gas limit for the next block in the sequence can be determined in two ways: Firstly, we can find the current block gas limit by running a local Ethereum Node and executing the `GASLIMIT OP CODE` on the previous mined block. This returns the block's own gas limit. There are many ways to return this data. For example, we can perform a Remote procedure call using the "JSON RPC API" such as `eth.getBlock("latest").gasLimit`, which returns the gas limit as an integer for the last block mined(18). Secondly, instead of having to rely on running a local Ethereum Node to use the gas analyzer, we can use a third-party service that runs multiple nodes for us. For our work, we have chosen to use EtherChain's API (13), as this is generally considered to be a reputable third party site that is an accurate representation of the current state of the Ethereum blockchain. EtherChain basis its numbers on the data returned from a variety of full nodes running across the world. For each run that is performed using the gas analyzer, we can perform an http request to the website which returns the current block gas limit. Running the equation above will give us the `minimum` block gas limit possible by the network.

There are some limitations with our analysis which leads to an over-estimation on the amount of gas for certain operations. As previously mentioned, the `SSTORE` operation has two different costs depending on the current value of storage. If `SSTORE` is performed on an empty storage value, changing this storage value to a non-zero value, the operation costs 20000 gas. If the operation is performed on a storage value that is non-zero (i.e there is some data stored at that 32-byte word), then the gas cost is lower at 5000. These are described in the three equations below(51):

$$G_{sset} = 20000 \quad (2.9)$$

$$G_{sreset} = 5000 \quad (2.10)$$

$$R_{sclear} = -15000 \quad (2.11)$$

Because static analysis cannot predict in advance if a storage value is non-zero or zero (as that requires dynamic analysis), we cannot determine if a `SSTORE` operation will cost 5000 or 20000. Since we are interested in the *upper bound*, we take that all `SSTORE` operations will consume 20000 gas, which assumes that the contract has no non-zero values in storage.

If storage is cleared (changed from a non-zero to a zero value - the default state) then 15000 gas is deducted from the total gas spent in the transaction, as this is added to the refund counter. Moreover, the `SELFDESTRUCT OP CODE`, which removes the contract and associated storage from the current state of the blockchain, adds 24000 gas to the refund counter. The refund counter of a transaction keeps track of how much gas to refund back to the caller `after` the EVM has finished execution - that is, when the op code `STOP` is executed. Hence, we do not need to incorporate it for our analysis. Refunds exist as data is cleared from the blockchain, reducing the storage size of the blockchain for all participants (50).

2.3 Exploits

2.4 Dynamic Loop Sizes

Loops in Solidity should be used sparingly as they are considered expensive. In a given transaction, a gas budget b is required that must be enough to start at s , the start node, iterate over the loop (and any statements in between) and continue to e , the end node. If the bound of a loop is determined by external user input or by the behaviour of an external contract, then there is a risk that a loop will be too large to cause an (out-of-gas) exception. Given that all data on the blockchain, including smart contracts, are considered “immutable” whose functionality is expected to last forever (or until the ‘suicide’ op code is executed - thereby disabling the smart contract’s functionality) we can assume that at some point in the future the bound will become too large for a given transaction to ever finish executing.

Solidity, “whose syntax is similar to that of JavaScript”(19), has syntax that can be arguably misleading. For example, the term `var` has a data type of `byte`, and is not a generic variable for which a value of any data type can be assigned to it unlike loosely typed languages such as Javascript. Instead, one can be fooled into assigning an integer to `var`, when instead they are assigning the number of bytes to it! Since the EVM stores bytes in 256-bit addresses, the length of `var` is at most 256. Therefore, any number greater than 256 will lead to an integer overflow(34).

For example:

```
uint causesOverflow = 512;

for (var i = 0; i < causesOverflow.length; i++){
    \ do stuff
}
```

will cause an integer overflow when i attempts to reach 257, but will instead overflow to zero again, and hence the loop will keep executing until an `out-of-gas` exception is thrown.

Loop bounds that are determined by user inputs are particularly dangerous. Because a contract owner does not know in advance how other contracts behave, the size of the loop may dynamically increase as a result of a malicious user, or by accident due to the growth of an array.

For example, suppose we have a list of accounts, where each account is represented as a `struct` consisting of a 160-bit contract or wallet address, and a positive balance pb where $0 \leq pb \leq 2^{256}$:

```
contract dynamicDividendPayments{
    struct Account {
        address a;
        uint balance;
    }
    Account[] accounts;
    function payDividends () returns (bool success) {
        for(var i = 0; i < accounts.length; i++) {
            uint tempBalance = accounts[i].balance;
            accounts[i].balance = 0;
            Account.address.send(tempBalance);
        }
    }
}
```

When the function `payDividends()` is called, it will loop over the list of structs, where the bound set by i is determined by the number of structs in the list (i.e the length of the array `accounts`).

On each iteration, the number of operations are performed in sequence: assigning the account balance to a temporary variable of type `uint`, the account balance is set to 0, and the previous account balance is sent to the address:

```
uint tempBalance = accounts[i].balance;
accounts[i].balance = 0;
Account.address.send(tempBalance);
```

These operations require a certain amount of gas to execute. Consequently, what happens when the number of accounts grow? More gas will be spent for the transaction by executing the operations `account.length` times. Eventually, the transaction will become too expensive and exceed the block-gas limit set by the network. It will never be mined, and the transaction will throw an error (`Error: Exceeds block gas limit`) in the web3js Javascript console.

Although one does not know the exact size of `accounts.length` when statically analyzing the contract above (because we do not know the exact data input when the transaction is executed) we can assume that should the loop bound by determined by user input, then the code is vulnerable to having an unbounded loop. For instance, a malicious user may repeatedly create y number of wallets programmatically, where y is a large number, and seed each account with just enough ETH to create an account on the contract above. If enough accounts are created such that the `for` loop will never cease execution before reaching the block-gas limit, then calling `payDividends()` will always fail. In the above example, `accounts.length` is determined by the number of addresses that are members of this contract. More specifically, they create their own `struct` when they join.

By assuming the worst case scenario that a user-defined loop bound will eventually lead to an unbounded loop, we can solve the problem in polynomial time as described in chapter 3, rather than having to iterate through the loop dynamically until all gas is exhausted or the loop terminates.

2.5 Wallet Griefing

When a call to an external address is made, the transaction's program execution counter is given to that external address. This gives rise to potential vulnerabilities in the smart contract, as the contract does not know exactly what computation will be done by that external address, and will only see the return values. Hence, a developer must write their contracts to manage any potential values returned.

Wallet Griefing occurs when a call to an external function inside a loop leads to an exception being thrown (for any reason), thus causing the whole transaction to fail even if other calls were successful.

For example, given the following code(46):

```
for (uint i=0; i<investors.length; i++) {
    if (investors[i].invested == min_investment) {
        // Refund, and check for failure.
        // This code looks benign but
        // will lock the entire contract
        // if attacked by a griefing wallet.
        if (!(investors[i].address
            .send(investors[i].dividendAmount)))
            {
                throw;
            }
        investors[i] = newInvestor;
    }
}
```

A "griefing wallet" could be a malicious contract which throws if its fallback function is called (a function that is always executed when an external address calls the contract with a blank function signature, that is, when no function is specified in the message call or transaction). In any Ethereum transaction or message call the first four bytes of the input data specifies what function will be called. If this is empty or is a call to an invalid function, the fallback function will still be called.

For example, given the following contract.

```
contract malicious{
    function (){
        throw;
    }
}
```

Whenever `investors[i].address.send()` is called, where `investors[i].address` is the address of `malicious`, an exception will be thrown, causing the transaction to always fail when it makes an external call to that address.

The `CALL`, `CALLCODE` and `DELEGATECALL` OP Codes represents three different methods of creating message calls. Recall that a message call is a form of **contract-to-contract** communication consisting of a number of key-value pairs, namely “they have a source, a target, data payload, Ether, gas and return data.”(19).

Because all three variations of a message call execute code from an external contract (the caller’s contract), we need to assume the worst-case scenario in that the external contract may contain a code executed via a message call whose execution path leads to an `out-of-gas` exception.

Even if a message call does not specify a function signature (i.e specify what function in the external contract it is calling, a message call with a blank function signature will still execute the fallback function as previously stated. If a message call does specify a function signature, the external contract’s function being called could still be modified over time, could call other functions within the contract itself or have some execution path that still leads to an `out-of-gas` exception, for example, if storage becomes too large and hence too expensive to retrieve.

2.6 Mass Clearing of Storage

Inexperienced developers may be tricked into thinking that relatively simple statements that are easy to write in Solidity and other languages are unlikely to cause major issues. However, this can simply be an abstraction in higher-level languages, when in fact many complex operations occur in the EVM. An example of this is the clearing of arrays or mappings in Ethereum.

In order to add data or change an existing value in storage, an `SSTORE` needs to be applied. Recall that `SSTORE` is an expensive operation consuming 20000 gas if adding to storage or 5000 gas if changing an existing value. Note that this is for *each* 32-byte word entry, not for the collection of words as a whole (for example in the form of an array).

Given an array of addresses of data type `address`, a user who wishes to clear all data in the array (i.e change multiple entries) could do the following:

```
addresses = new address[] (0);
```

In many languages such as Java this is a relatively simple operation, as the Java Virtual Machine (JVM) will free that memory used without any issue, assuming the array does not contain an enormous number of elements. However, in smart contracts, the EVM will loop over each entry in `addresses` and apply the `SSTORE` operation in each iteration, setting each value to zero. Note that even if the existing value is zero, the operation will still cost 5000 gas, as the value is being set to zero again.

Assuming a block gas limit of 6.7 million gas, and given that each transaction costs 21000, the number of elements that can be cleared in a single transaction is roughly:

$$entriesCleared = \lfloor (6700000 - 21000) / 5000 \rfloor = 1335 \quad (2.12)$$

Hence, assuming no other operations take place in the transaction, only 1335 entries can be cleared in a single transaction. If `addresses` holds more than 1335 elements, then `addresses = new address[] (0)` will never execute!

A more gas efficient and safer method is to not delete elements at all (thus avoiding the use of `SSTORE`) but instead keep track of what entries are "in use" and a counter for the number of these entries(23). Although this method is inefficient for the blockchain (as the blockchain needs to keep maintain storage for all elements not in use), it is more gas efficient as unused elements can simply be ignored.

Thus, we check for instances where `SSTORE` operations are performed within a loop, as this is an inefficient coding pattern that can easily lead to an `out-of-gas` exception.

2.7 Summary

In this chapter we introduced three vulnerabilities that have significant consequences for smart contracts. The Wallet Griefing attack allows malicious contracts to cause a loop to always fail simply by throwing an exception. Having a dynamically bounded array can also cause the loop to never complete by consuming too much gas, as the loop bound grows over time as more data is added to the smart contract.

Moreover, while clearing data structures may seem like a trivial matter in other programming languages, doing so in smart contracts is expensive as clearing storage leads to a change in persistent storage on the blockchain, which all other nodes must replicate. Given the seriousness of these vulnerabilities, how can we automatically detect them?

CHAPTER 3

Analysis

In this chapter we will present an analysis to identify the three aforementioned resource-based vulnerabilities of smart-contracts. To identify these vulnerabilities we employ techniques developed in static-program analysis (32) that is concerned with abstracting the real semantics of a program making the detection of vulnerabilities decidable. There are various ways to express static program analysis. In our approach, we provide a mix of techniques. We introduce a frequency analysis that estimates the maximum frequency count for a statement based on a given gas budget, thus determining whether statements are cyclic or not if the maximum frequency count is greater than one.

Finally, we provide logical specifications that use the results of the maximum frequency analysis. These specifications characterize the resource-based vulnerabilities. The specifications are approximations, i.e., they cannot detect with complete certainty whether a vulnerability may become an exploit. They just indicate whether there is the *potential* for an exploit to exist.

Although these techniques were integrated into the Vandal framework as described below, the algorithm used to calculate maximum frequencies is agnostic to any particular framework, if given a control-flow graph and a gas budget. However, the logical specifications used to identify vulnerabilities is specific to the Datalog programming language in the context of the Soufflé engine. Nevertheless, the first-order logic rules can be adapted for any other logic programming language and framework.

To analyze potential vulnerabilities in smart contracts as a result of exceeding the gas limit, it is necessary to derive an effective way of analyzing the costs of executing a particular sequence of `OP CODES` contained within a given execution paths for all possible paths. Currently, there is no efficient way of doing so. `web3js`, a JavaScript-based RPC Application Programming Interface (API) has a particular function called `web3.eth.estimateGas(tx)`, which executes a simulated transaction to the blockchain and returns the estimated gas consumed as if the transaction was actually sent. However,

it makes several assumptions, including that no exceptions are thrown and all blocks are mined in the same manner.

The Solidity compiler can estimate gas using the command `solc --gas <solidity file>`. However, it makes the same assumptions as `web3.eth.estimateGas(tx)`. Both methods do not give any insights into the gas spent at specific points in the execution of a contract or for any particular execution path.

Existing formal verification tools such as "Oyente" and "Gasper" as described in Chapter 7 use symbolic execution to conduct their analysis. Symbolic execution is a static analysis technique where each possible execution path is checked for a given input x . This is a slow process that is exponential in the number of computational steps required. Luu *et al* (2017) were only able to execute on average nineteen paths in each contract. In many relatively complex contracts, this is far from the total number of paths.

We propose to use two methods to analyze gas costs across execution paths which are more efficient than symbolic execution. Our initial algorithm uses an integer linear programming solver which takes as input a sequence of basic blocks and their edges and calculates the maximum frequency of each edge to determine the path that leads to the greatest gas cost. Although still requiring an exponential number of steps, there are far fewer paths that need to be analyzed compared to symbolic execution.

The integer programming method will be the benchmark compared to the more optimal, iterative solution using Dijkstra's Algorithm that runs in polynomial time. We demonstrate that it is orders of magnitude faster than solving maximum frequencies using integer linear programming for large, non-trivial contracts.

This analysis helps us determine the logic specifications to identify a number of exploits that have not been covered in previous papers. These exploits often lead to `out-of-gas` exceptions or a denial-of-service problems, and include the identification of dynamically bound loops, which can grow in size over time making a transaction more expensive to execute, the Wallet Griefing attack, which can deny access to a contract, and the mass clearing of storage for data structures such as mappings and arrays, which is a common error made by developers that is an expensive operation to run.

3.1 General framework

The detection of the exploits mentioned in Chapter 2 extends an existing framework. As described in Figure 3.1, it consists of a decompiler that transforms low-level bytecode generated by the EVM to its three-address code (TAC) representations, as well as a series of logical specification in Soufflé that analyze the results from the decompiler.

In order to perform our analysis on the blockchain, the following steps are performed:

- (1) A scraper written in Javascript uses an Ethereum node via the Parity software client to scrape blocks from the local blockchain that is kept synced with other Ethereum Mainnet blockchains using JSON-RPC calls. This produces, amongst other data, the contract bytecode of each contract stored in a particular block in the following format: `<contract_hash_runtime>.hex`. These files are stored in a directory. Each file contains the contract bytecode stored on the blockchain.
- (2) A disassembler runs the `<contract_hash_runtime>.hex` file to translate contract bytecode to a series of `OP CODES` mnemonics and their impact on the EVM stack and its program counter.
- (3) The decompiler is run on the disassembled output to produce a series of basic blocks and three address code (TAC) output. This is achieved by simulating the execution of the EVM. The TAC output is then segregated into a set of basic blocks that are then connected to one another by creating a list of predecessor and successor for each block, thus form a CFG. While building the CFG, we analyze the gas consumed per block by iterating over the each `OP CODE` that is executed and its associated costs.
- (4) After the CFG is created, we perform our maximum frequency analysis by first transforming each vertex (which represents a basic block) to a weighted edge as described in Algorithm 1, where the weight is the total gas consumed by that basic block. The maximum frequencies of each edge are then calculated using the iterative solution as described in Algorithm 2 or using the Integer Linear Programming (ILP) solution to determine what statements exists within a loop and the number of times those statements can be vided for a given gas budget b . This produces a new CFG where each edge has an associated maximum frequency based on b .
- (5) A series of `.fact` files are produced and exported by the decompiler. Facts are statements that are assumed to be true and form the basis of all predicates constructed in our logical analysis.

These facts track the flow of data in the program, the statements themselves, the use of storage and memory, graph dominance (described below) and the start and end statements of all cycles found in the program (given *b*). These files are essential for our logical specifications to automatically detect vulnerabilities.

- (6) Soufflé is run using the `.facts` files given combined with our datalog programs that identify patterns in the program mapped to a given set of vulnerabilities. If a pattern is identified, Soufflé will flag the program as containing at vulnerabilities and which statements have it. If a given pattern is not identified, then nothing will be flagged. A program either has a given vulnerability or not. Note that a variety of patterns can be coded in datalog, not just to identify vulnerabilities but anything in particular, for example, inefficient coding or gas consumption patterns.

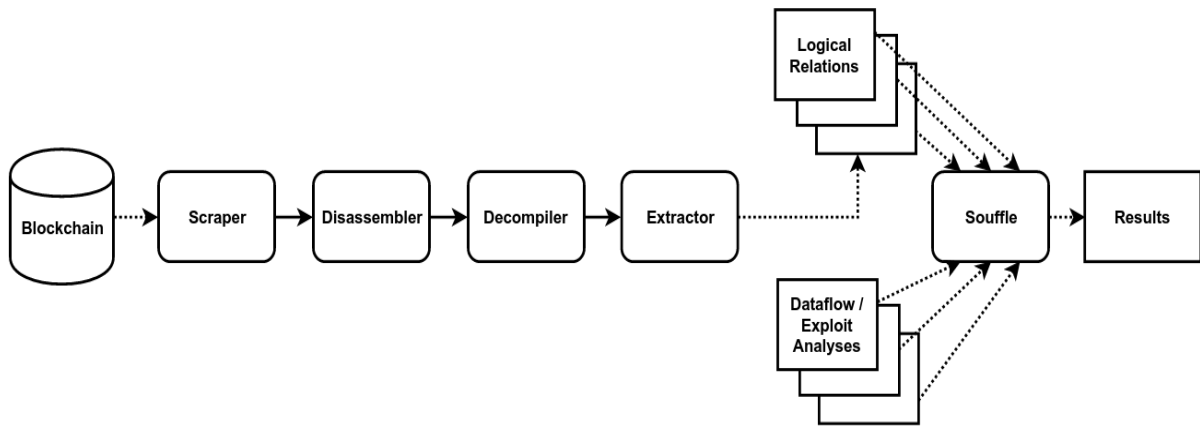


FIGURE 3.1: Security Analysis Pipeline

3.2 Loops

Loop analysis is a difficult task since high-level loops in Solidity need to be analyzed at the EVM level. The only way to deduce loops is to analyse the underlying CFG of a smart-contract. The Vandal system provides the functionality of producing control flow-graphs of smart contracts. However, the Vandal system has to be extended to find loops for resource-based vulnerabilities of smart contracts. This loop analysis focuses on sub-graphs that form cycles, and is detected by determining the maximum frequency of each edge.

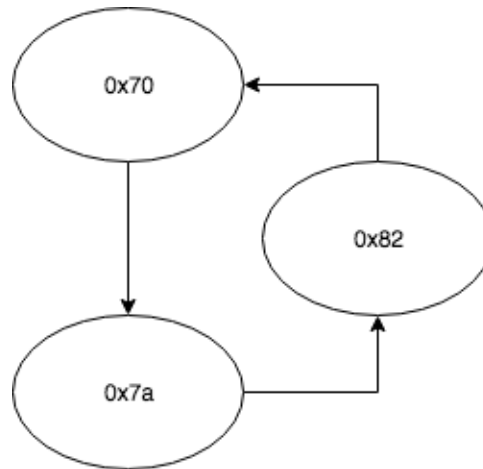


FIGURE 3.2: The control flow graph of a simple function with a loop

A loop that has a fixed number of iterations needs to be executed before that transaction finishes and is submitted to the blockchain. If the amount of gas forwarded for the transaction exhausts before the loop concludes, then the transaction will revert. These loops are easier to analyze than dynamically bounded loops because there is a known number of iterations at run-time.

The example below contains a kernel of source code called contract `safeLoop` with a function `test()` that contains a fixed number of iterations 256:

```

contract safeLoop{
    function test() constant returns (uint) {
        uint x = 0;
        for (uint i= 0; i<256; i++){
            x++;
        }
        return x;
    }
}
  
```

This produces the following cyclic subgraph as depicted in Figure 3.2. The basic blocks labelled `0x70`, `0x7a` and `0x82` contain the `for` loop functionality mentioned above. Block `0x70`, which strictly dominates all statements within the cycle given in figure 3.2, contains a comparison check against the loop bound. Note that block `0x70` only has one outgoing edge to block `0x7a` as the function will return

`x` once the loop has concluded, and does not perform any other operations after that. The TAC output of figure 0x70 is given below:

```
Block 0x70
0x75: V29 = LT {0x0, 0x1} 0x100
0x76: V30 = ISZERO 0x1
---
Block 0x7a
0x7c: V32 = 0x1
0x7e: V33 = ADD 0x1 {0x0, 0x1}
---
Block 0x82
0x87: V35 = ADD 0x1 {0x0, 0x1}
0x8d: JUMP 0x70
```

The crucial statement above is `0x75: V29 = LT 0x0, 0x1 0x100` as this applies a less-than comparison against the hex value `0x100`, with the value 256 in base 10 in order to check against the bound of the loop. Block `0x7a` performs the change to the loop's counter (in this case, by incrementing `i`, and Block `0x82` performs operations within the `for` loop, which increments the variable `x`

In contrast, a loop with a fixed number of iterations 256^3 will exceed the block-gas limit:

```
contract unsafeLoop{
    function test() constant returns (uint) {
        uint x = 0;
        for (uint i= 0; i<256**3; i++){
            x = x*i + x;
        }
        return x;
    }
}
```

There are also loops whose bounds are determined by variables that can change over time (i.e dynamically bounded loops). For instance, they may depend on user input or some value returned from storage using `SLOAD`.

As previously mentioned, loops determined by user input could grow dynamically such that iterating through the loop exceeds the block gas limit or becomes too economically expensive to run such an action. This will lead to a “Denial of Service” for all transactions that must attempt to iterate the loop.

Consider the contract below (10):

```
pragma solidity ^0.4.11;
contract dynamicLooping{
    struct Payee {
        address addr;
        uint256 value;
    }
    Payee [] payees;
    function payOut() returns (uint) {
        uint x = 0;
        for (uint i= 0; i<payees.length; i++){
            x++;
        }
        return x;
    }
}
```

A user is able to become a `Payee`, which consists of both an address and a certain ‘`uint256 value`’, which could represent, say, the balance in `Wei` that is held by that particular address.

If an address (such as the address of the contract owner) were to call the function `payOut ()`, a payment in `Wei` will be sent out to each address according to the balance the user holds. However, because the loop is bounded by `payees.length`, and the number of `payees` may always increase, the contract will reach a point where executing the function:

```
payees[i].addr.send(payees[i].value);
```

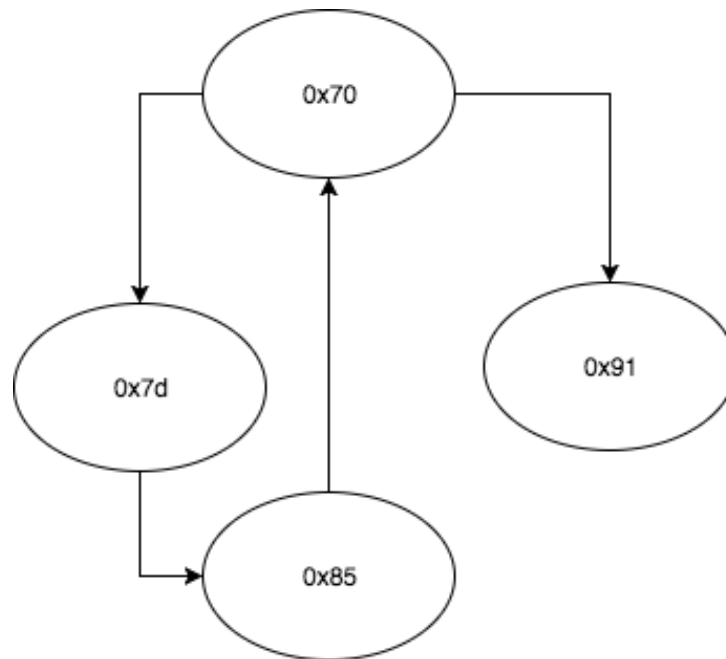


FIGURE 3.3: The control flow graph of a simple function with a loop bound determined by external input.

will exceed the block gas limit, as `payees.length` can increase in length over time as more and more addresses are added to `payees`.

Hence, our security analysis should determine if a particular loop bound can be dynamically increased over time.

The Contract `dynamicLooping` produces the following cyclic subgraph in 3.3.

The blocks that contain the `for` loop are `0x70`, `0x7d` and `0x85`. These three blocks have the following TAC output:

Block `0x70`

`0x70: JUMPDEST`

`0x71: V28 = 0x0`

`0x74: V29 = S[0x0]`

`0x78: V30 = LT {0x0, 0x1} V29`

`0x79: V31 = ISZERO V30`

`0x7a: V32 = 0x91`

`0x7c: JUMPI 0x91 V31`

```

---
Block 0x7d
0x7f: V33 = 0x1
0x81: V34 = ADD 0x1 {0x0, 0x1}
---
Block 0x85
0x85: JUMPDEST
0x88: V35 = 0x1
0x8a: V36 = ADD 0x1 {0x0, 0x1}
0x8e: V37 = 0x70
0x90: JUMP 0x70

```

The output is, not surprisingly, very similar to the output of `safeLoop` in that block `0x7d` performs the operations the `for` loop, in this case by simply incrementing a variable. However, the key difference is the **loop bound**, which in `dynamicLooping` is set in the statement:

```
0x74: V29 = S[0x0]
```

which is then compared to `i` to ensure it is less than the number returned by `S[0x0]`, where `S[0x0]` represents the value of the word loaded from storage located at `0x0`. A word is a piece of data represented by 32-byte and is stored in a mapping of key-value pairs of 32-byte words(19) This is exactly the size of `payees.length`, where `payees` is a `struct` stored using the `SSTORE OP` CODE.

Hence, we can conclude that a loop bound whose value is determined by a word retrieved from storage is essentially a word of dynamic length, and hence represents a loop bound whose value could grow over time as more data is added and the word length increases.

This analysis can be conducted simply by look at the node at the beginning of the cycle.

In the above example, `0x59` is the node that begins the loop by first checking if the conditions within the loop bound are being met. In the case that the check evaluates to `false`, then the loop will break, and the execution path will continue outside of the cycle. If the check evaluates to `true`, it will continue along the cycle path and repeat back to `0x59`.

3.3 Wallet Griefing

Recall that a contract is said to be vulnerable to the “Wallet Griefing” exploit if a message call (defined by the use of a CALL, CALLCODE or DELEGATECALL exists within a loop, as the external contract being called may perform some operations that throws an exception, thus prevent the loop from ever completing.

An example of Wallet Griefing in depth is below(46):

```
contract walletGriefing {
    mapping (address => uint256) balances;
    function grief () {
        for (uint i=0; i<balances.length; i++) {
            // Refund, and check for failure.
            // This code looks benign but will lock the entire contract
            // if attacked by a griefing wallet.
            if (!(investors[i].address.send(balance[i])))
            {
                throw;
            }
            investors[i] = newInvestor;
        }
    }
}
```

This produces a CFG with the following cyclic subgraph as depicted in Figure 3.4.

The key block to analyze in the cycle is block 0xe4. This block contains the operations where the message call is performed and the return value checked:

```
0xf1: V70 = M[0x40]
...
0xf6: V73 = M[0x40]
0xf9: V74 = SUB V70 V73
0xfe: V75 = CALL V68 V48 V66 V73 V74 V73 0x0
```

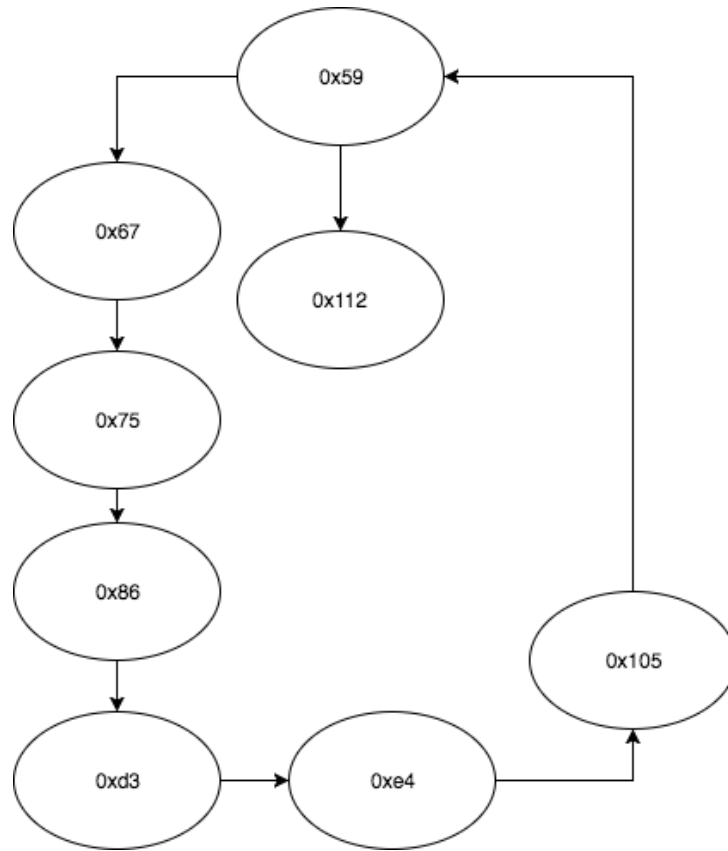


FIGURE 3.4: The control flow graph of a simple function with a message call inside the loop

Where `v75` contains the return value of the call. Note that using the `send()` or `transfer()` functions in Solidity will cause 2300 gas to be forwarded to the address being called. If `call.value()` were used instead, then the gas forwarded would be the remaining gas left in the transaction returned by the `GAS` operation.

If the address being called causes an exception to be thrown then `v75` will never contain the return value of the call (since a return was not made) and the EVM will jump to an invalid destination. Because one does not know the value of `v75` in advance, or that it will return anything at all, `v75 = CALL v68 v48 v66 v73 v74 v73 0x0` is a vulnerable statement that could cause the loop to never finish executing.

Hence, if a `CALL`, `CALLCODE`, or `DELEGATECALL` is made within a loop (i.e within a subgraph that is cyclic), then we can conclude that the contract is vulnerable to Wallet Griefing.

3.4 Mass Clearing of Storage

Recall that clearing storage is an expensive operation that should be minimized whenever possible. There are several ways a developer might attempt to clear storage in Solidity. One method is to write a loop bounded by the size of the array, and reassign each value to some zero value:

```
contract clearStorage {
    address[] public addresses;
    function testClearStorage() {
        //Iterate over each element and clear storage
        for (uint i = 0; i<addresses.length; i++){
            //set to zero value
            addresses[i] = 0;
        }
    }
}
```

However, an even more simple way of clearing storage is shown below:

```
contract clearStorage {
    address[] public addresses;
    function testClearStorage() {
        addresses = new address[] (0);
    }
}
```

When initiated, the contract contains an array of type `address`. Calling the function `testClearStorage()` causes the smart contract to attempt to clear all the `address` elements in `addresses` and set them to some non-zero value. Note that because the EVM is single-threaded, each element must be set to zero itself on an iterative basis.

Although `clearStorage` appears to be relatively simple, it produces a CFG consisting of 34 basic blocks, and produces the cyclic subgraph C :

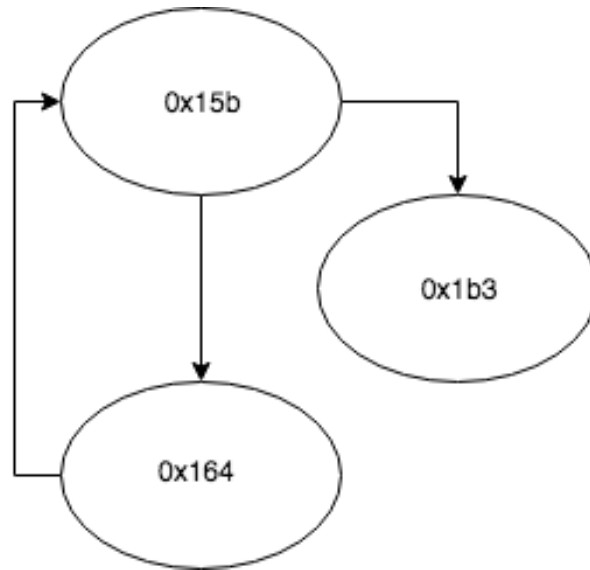


FIGURE 3.5: *C* allowing a user to clear an array of addresses

In Block $0x164$, the element returned (which is always a 32-byte word) first checked to see if it has already been assigned a zero-value. If a zero-value has already been assigned, then the reassignment is skipped, thus representing an optimization made by the EVM. If the value is non-zero, then an `SSTORE` operation is performed, setting it to zero. This is described in the TAC below:

```

V97 = S[S1]
...
V104 = MUL V103 0x1
V105 = OR V104 V101
S[S1] = V105

```

Note that a comparison check is made in Block $0x15b$ against the length of the the collection of elements in that particular data structure, whether it is a mapping or an array. This check is show below:

```

V90 = GT V89 S2
V91 = ISZERO V90
V92 = 0x1b3
JUMPI 0x1b3 V91

```

`S2` is a variable containing the length of the data structure. Note also that the `SLOAD` operation to attain the length value is performed only once. The value used by `JUMP` determines where in the control flow graph the execution will go to. If the check returns false, it will exit the loop to block `0x1b3`, else it will once again jump to block `0x164`.

3.5 Maximum Frequency Analysis

In order to detect the exploits mentioned in chapter 2, the decompiler needs to be able to detect the presence of loops in a given CFG. Chen *et al* (2017) ignore all loops and hence does not even parse them when given a smart contract. To determine if loops exist in a contract, and if a particular sequence of statements falls within a loop, we need to calculate the maximum frequency of each edge - the number of times that edge is visited given an execution path and a set gas budget. If the maximum frequency is greater than one, then we know that statement with that incoming edge is inside a loop.

A CFG consist of nodes that represent a given amount of computation and edges that connect these nodes together to creates paths that could be traversed when executing the program.

Recall that the total gas cost cannot exceed the block gas limit, which can be determined by looking at the gas cost of the previous block and taking that as the block gas limit. This limit cannot be exceeded by any path and a user cannot set a gas budget greater than it. Note that the block gas limit may vary slightly than the previous block as described in Chapter 2.

Hence, we have the following:

- (1) Let t , b and u be the gas cost, the total gas budget and block gas limit respectively ($b \leq u$).
- (2) Code the costs of each `OP_CODE` according to the formulae specified in *Appendix G. Fee Schedule*, page 20. In the Vandal framework, this was coded in as a parameter for each `OP_CODE` in `Op_Codes.py`, where each `OP_Code` is a tuple consisting of its symbol (for example, `ADD`), the impact on the stack of performing the operation, and its associated gas cost.
- (3) When running the decompiler, determine the gas consumed per block based on the results in 2. Take note of the successors of each block (which represent an edge to that block in an execution path).

This produces a control flow graph $G(V, E)$ where V is the set of basic blocks (vertices) and E is the set of edges (successors).

A CFG consists of nodes that represent a given amount of computation and edges that connect these nodes together to create paths that could be traversed when executing the program.

We can now calculate the maximum frequency of each edge for a given control flow graph. The algorithm below is shown to run in polynomial time. The maximum frequency of a given edge $e(v1, v2, g)$, where g is the total number of gas consumed by traversing this edge (i.e the weight of the edge), $v1$ is the outgoing vertex of e and $v2$ is the incoming vertex of e , is defined to be the number of times a visitor in a given path p can visit the node with a given gas budget b . The purpose of calculating maximum frequencies is to determine the maximum number of iterations for each edge for a given transaction and its gas budget b .

3.6 Transforming the CFG

In order to perform the maximum frequency analysis, we first need to create a new CFG where all basic block gas costs (denoted as “weights”) become edges between the basic blocks to create a connected graph C , where there exists a common start and end node s and e respectively that is connected to all “real” start and end nodes respectively. This connected graph is denoted by $C(V, E, s, e)$, where s and e are the start nodes respectively, and V and E are the set of vertices and edges respectively. Both s and e have outgoing and incoming edges with a weight of zero (since these do not represent actual computation that takes place in the EVM) respectively. Each basic block with no predecessors or successors will be connected to s and e respectively.

To produce C , the following steps are taken:

- (1) For each basic block, create an edge $e(v1, v2, g)$, where $v1$ is the outgoing vertex, $v2$ is the incoming vertex, and g is the gas consumed traversing that edge (i.e the edge weight).
- (2) Create a start node s .
- (3) Create an end node e .
- (4) If $v1$ does not have a predecessor vertex v , create a new edge of zero weight from s to $v1$ (i.e $e(s, v1, 0)$).
- (5) If $v2$ does not have a successor vertex v , create a new edge of zero weight from $v2$ to e (i.e $e(v2, e, 0)$).

Note that all of the gas consumed of a basic block goes to the incoming edge of v .

Algorithm 1: Constructing the new CFG

Data: $cfg.blocks$

Result: A new cfg_{ncfg}

for all blocks n in $cfg.blocks$ **do**

if no predecessors **then**

 | add incoming edge from s

else

for all predecessors p in n **do**

 | add incoming edge from p with weight equaling the gas consumed of n .

end

end

if no successors **then**

 | add outgoing edge to e with weight 0

else

end

3.7 Iterative Solution

Calculating shortest paths to determine what is the maximum number of times the edge can be visited which represents the upper bound of iterations, as there are no better execution paths that would comprise of a cheaper gas cost. This is opposed to calculating the longest path, which would determine the minimum frequency of a node and represent the lower bound of the number of edges or statements in a transaction. Such a problem is NP-Hard given a directed cyclic graph which can exist in the CFGs of Ethereum smart contracts due to its Turing-completeness. Furthermore, calculating longest path may assume that vertex is not reachable when indeed it is based on a shorter execution path that the EVM can traverse. Although cycles can be detected using a variety of algorithms such as Depth-First Search by detecting back-edges (an out-going edge for vertex u which visits a vertex v that has a path consisting of an incoming edge to u), these algorithms do not work for a given resource constraint (i.e a gas budget). Moreover, these algorithms can only detect cycles and not the number of iterations of each cycle for a given budget. Hence, the maximum frequency algorithm described here is significant given the behaviour of gas, how miners behave, and the EVM.

Note that we assume no negative cycles exist, which is a safe assumption as negative gas costs do not exist in the EVM. If negative gas costs did exist, that would imply that the network would be paying you to run your analysis!

Given the Directed Connected Graph C and the following paths in C , where s and e are the start and end nodes of C respectively:

$$p(s, p), p(p, q), p(q, e)$$

To find the maximum frequency of $e(p, q)$, for each p and q , run Dijkstra's algorithm on $p(s, p)$ and $p(q, e)$ to calculate the total gas consumed in the shortest paths $sp(s, p)$ and $sp(q, e)$, as described in Algorithm 2.

Now, consider the following three cases:

- (1) If no path $p(s, p, q, e)$ exists, then:

$$Pmax = 0$$

- (2) A path $p(s, p, q, e)$ exists, and there is no cycle (i.e a shortest path $sp(q, p)$), then

$$Pmax = \begin{cases} 1, & \text{if } sp(s, p) + e(p, q) + sp(q, e) \leq b \\ 0, & \text{otherwise} \end{cases}$$

- (3) If $p(s, p, q, e)$ and $sp(q, p)$ exists, a cycle exists for p , then:

$$mf = \lfloor [sp(s, p) - e(p, q) - sp(q, e)] / (e(p, q) + sp(q, p)) \rfloor$$

$$Pmax = \begin{cases} 1+mf, & \text{if } sp(s, p) + e(p, q) + sp(q, e) \leq b \\ 0, & \text{otherwise} \end{cases}$$

If $p(s, p, q, e)$ does not exist, then p and q will never be visited in a given transaction, as a transaction must always start at s and end at e . Hence, the maximum frequency must be zero. In the second case, $e(p, q)$ can only be visited at most once in a given transaction, as no cycle exists. Thus, if $e(p, q)$ can be visited from s to e with a cost less than or equal to b , then it will be visited once, and the maximum frequency set to one. Otherwise, $e(p, q)$ will never be visited for that b and hence the maximum frequency will be set to zero. In the third case, the maximum frequency, if it passes the condition

as mentioned in case two (i.e that $sp(s, p) + e(p, q) + sp(q, e) \leq b$), the maximum frequency will be at least one. However, the maximum frequency will also need to include the number of cycles, where each cycle adds one to the maximum frequency count. This can be easily determined by taking $sp(s, p) - e(p, q) - sp(q, e)$, the cost of $p(s, p, q, e)$ given one iteration, and dividing it by the cost of running a cycle (that is, $e(p, q) + sp(q, p)$).

Performing the maximum frequency calculations according to the formulae above is described in Algorithm 2. Note that Dijkstra's means "Dijkstra's shortest path algorithm" (49), which utilizes priority queues to improve performance:

Algorithm 2: Calculate Maximum Frequencies

```

for  $e$  in  $C$  do
     $sp\_s\_p \leftarrow dijstrakas(s, p)$ 
     $sp\_q\_e \leftarrow dijstrakas(q, e)$ 
     $isCycle \leftarrow dijstrakas(e(V(1)), e(V(0)))$ 
    if no path from  $s$  to  $p$ ,  $p$  to  $q$ ,  $q$  to  $e$  then
        |  $mf \leftarrow 0$ 
    else if  $[sp\_s\_p + w(p, q) + sp_{qe}] \leq b$  then
        | if  $isCycle$  then
            | |  $mf(e) \leftarrow \lfloor [b - sp(s, p) - e(p, q) - sp(q, e)] / (e(p, q) + sp(q, p)) \rfloor$  else
            | | else
            | | |  $mf \leftarrow 1$ 
            | | end
            | end
        | else
        | |  $mf \leftarrow 0$ 
    end

```

The resulting algorithm can be shown to have polynomial complexity.

CLAIM: Algorithm2 runs in $O(E((n + e) \log n))$.

PROOF: The number of times the maximum frequencies needs to be calculated corresponds to E . For $e \in E$, Let p, q be the vertices in $e(p, q)$, and s and e be the start and end nodes for the connected graph C respectively. Dijkstra's algorithm needs to run three times, one to determine, the shortest path from

s to p , another from q to e and one to check if a shortest path cycle exists between p and q . Dijkstra's algorithm utilizing priority queues runs in $O((n + e) \log n)$ time, where n is the number of vertices in C . Since there are E number of iterations required, the complexity is $O(E((n + e) \log n))$, which is polynomial.

3.8 Integer Linear Solution

The maximum frequency problem can also be described as an integer linear optimization problem. The objective is to maximize the edge frequency for each edge (p, q) defined as $f(p, q)$ for all edges e in $G = (V, E)$. This problem is subject to five constraints. The sum of the edge frequencies between the start node s to the node u must equal one, where u is in the set of successor nodes of s (i.e there must exist a path p from the start node s to the end node e). The sum of the edge frequencies between the node v and the end node e must equal one, where v is in the set of predecessor nodes of e . The sum of the frequencies going from u to the v must equal the sum of the frequencies going out of v to w for all nodes v in V not including s and e . The sum of the gas consumed for a given edge (u, v) multiplied by the edge frequency (u, v) must be less than or equal to the gas budget b , where (u, v) is an edge in the set of edges E . Finally, b must be less than or equal to the block gas limit L and b must be equal to or greater than 21000 (the minimum cost of a transaction)(43):

$$\text{maximise} \quad f_{pq} \quad (3.1)$$

$$\text{subject to} \quad \sum_{u \in \delta^+(s)} f_{su} = 1 \quad (3.2)$$

$$\sum_{u \in \delta^+(v)} f_{uv} = \sum_{w \in \delta^-(v)} f_{vw} \quad \forall v \in V \setminus \{s, e\} \quad (3.3)$$

$$\sum_{u \in \delta^-(e)} f_{ue} = 1 \quad (3.4)$$

$$\sum_{(u,v) \in E} g(u, v) f_{uv} \leq b \quad (3.5)$$

$$21000 \leq b \leq L \quad (3.6)$$

$$\text{and} \quad f_{uv} \in \mathbb{Z}_0^+ \quad \forall (u, v) \in E \quad (3.7)$$

All Integer Linear Program are NP-Hard unless P=NP. As a result, we expected this solution to run far lower than the Polynomial solution described in Chapter 4, especially since the maximum frequency calculations need to be performed for all $e \in E$.

3.9 Logic of Resource Exploits

The logic specifications to detect the vulnerabilities explored in Chapter 2 were written in Datalog, a logic programming language using the Soufflé framework(42). Running the Vandal decompiler produces a series of `.fact` files which are a collection of statements assumed to be true(42). These files include the list of statements, edges between statements, what variables are used by what statements, cyclic statements, and the start and end statements of the CFG. Common rules across all vulnerability detection scripts were put in place to track the dataflow, paths between statements, controlling statements and graph dominance. A vertex v is set to dominate another vertex u iff all paths $P(s..u)$ must go v . Graph dominance, important for determining which statements control others, for every vertex for all paths in the CFG is calculated after the CFG is created in Vandal using the networkx(31) library, which utilizes an algorithm developed by Cooper, Harvey and Kennedy (1990) that runs in $O(n^2)$ (11). These rules and facts are essential for establishing the predicates below.

Given a particular control-flow graph $G(V, E)$, a cycle is considered to have a variable loop bound depended on user input for a given basic block b if the following conditions are met:

- (1) A cycle exist is the set $V(u..v)$, where $V(u..v)$ are vertices within the cycle.
- (2) b is at the beginning of a cycle.
- (3) b is a controlling statement with a fork (i.e two successor nodes or outgoing edges), where on one successor is cyclic, the other successor is acyclic.
- (4) There is one incoming edge from one $V(u..v)$.
- (5) a LT ("Less than") or GT("Greater than") check is made against a variable whose value is retrieve from storage using `SLOAD` or `MLOAD`. `SLOAD` loads a word from storage, while `MLOAD` loads a word from memory. Note that is is unlikely, but still possible, that user behaviour may lead to a significant expansion in memory when executing a smart contract. However, it is likely to lead to an out-of-gas exception before memory becomes big enough to cause an out-of-gas exception within the loop itself.

Employing these rules produces the following predicate as shown in Figure 3.6. This predicate states that a statement within a cycle (denoted by `freq(x)` that represents all beginning and end statements for a cycle) dominates the return value of an `SLOAD` operation that uses an LT or GT check against some variable. If the predicate holds, we can state that a dynamic bound exists for a cycle in that smart contract. Note that the analysis *may* include instances where data is loaded from storage (`SLOAD`)

```

1 dynamicBound(x) :- freq(x),
2                   sloadMloadResult(resVar, sloadMloadStmt), imdom(x,sloadMloadStmt),
3                   depends(cond, resVar),
4                   use( cond, LTGT),
5                   isLForGT(LTGT).
6 )))))))

```

FIGURE 3.6: Datalog Program: Specification for dynamic bound.

or from memory (MLOAD) and a comparison check is made against this, although this is also a gas inefficient coding pattern as stated by Chen *et al*(2016) (6).

The "Wallet Griefing" and "Mass Storage Reassignment" exploits can be detected by performing simple taint analysis, the process of marking data as being "tainted" should it originate from an untrusted source(7). In this case, message calls to an external contract should be marked as tainted because the execution path of the contract is unknown in advance.

Given a particular control-flow graph $G(V, E)$, the following execution paths exist such that:

- (1) The statement is a controlling statement.
- (2) A cycle exist is the set $V(u..v)$, where $V(u..v)$ are vertices within the cycle.
- (3) A fork exists, where there exists two successors. One successor is cyclic, the other successor is acyclic.
- (4) A CALL, CALLCODE, DELEGATECALL exists within $u..v$.

Any execution after one of those three OP Codes is assumed to be strictly tainted, because if the computation in the external contracts leads to an exception, the entire transaction will be rolled back, and no state changes will have been made.

Employing the above rules creates the following predicate:

```

1 checkedWalletGriefing(x) :- freq(x), path(callStmt, x), path(x, callStmt), callStmt(callStmt).
2 )))))))

```

Which states that a CALL, CALLCODE or DELEGATECALL statement is cyclic as a path exists from x to a message call statement, and a path exists between the message call and x. This property is enough to detect Wallet Griefing, as the mere presence of a message call, regardless of the payload data or

execution, makes the smart contract vulnerable as the external contract being called may throw an exception, causing the loop to never complete.

A very similar predicate is also established when checking if `SSTORE` exists within a cycle:

```
1 checkedMassSSTORE(x) :- freq(x), path(sstoreStmt, x), path(x, sstoreStmt), sstoreStmt(sstoreStmt).  
2 ))))
```

An `SSTORE` statement that exists within a loop will be executed is an inefficient coding pattern, as discussed in chapter 2 and argued by Chen *et al* (2017). The `SSTORE` statement will be executed a given number of times as determined by the maximum frequency of the given edge where the operation is performed. The type of data being saved is largely irrelevant as the lowest gas cost is 5000 (when setting a non-zero value to a zero value in storage), and as previously discussed in chapter 2, there are more gas efficient coding patterns that should be used.

Experiments

In conducting our experiments we had following questions:

- (Q1) How many vulnerabilities can be found on the Ethereum Mainnet blockchain for each of the three vulnerabilities analyzed?
- (Q2) How long does the analysis take (for Souffle and calculating maximum frequencies)?
- (Q3) What percentage of edges have a maximum frequency greater than one? (i.e are statements within a loop).
- (Q4) What is the performance comparison of the iterative solution versus the integer linear program?

Three experiments were conducted based on the maximum frequency algorithms and datalog specifications covered in chapter 3. To test the maximum frequency calculations of both the iterative solution and integer linear programming, the algorithms were written into the existing decompiler software written in Python. The final experiment was running our datalog specifications on the three classes of vulnerabilities discussed in chapter 3 on all successfully decompiled contracts. These experiments were conducted on all contracts scraped from the Ethereum Mainnet blockchain as of 28 July 2017, totaling 886,323. These contracts were scraped using a scraper built in NodeJS in order to run the analysis for the initial Vandal framework previously mentioned (1). The scraper performs JSON-RPC calls to the blockchain using a Parity node and stored in PostgreSQL. The 28 July 2017 scrape was considered appropriate due to the large number of smart contracts for which we could test on. Performing our analysis on a more recent scrape would have required significantly more time and/or computing power, as the number of smart contracts is over two million as of October 2017 (16). Moreover, 886,323 contracts is arguably a large enough sample size to draw important inferences from in our results.

All results were generated using an Ubuntu 16.04.02 LTS, x86_64 64-bit machine with 24 CPUs, with 2 threads per core. Each CPU is an Intel(R) Xeon(R) x5650 model with 2.67GHz processor. No over-clocking was used.

To perform the analysis, an existing scripted called `analyse.py` was modified and used that allowed us to run the iterative maximum frequency calculations first, and use those results to perform our Soufflé analysis on each contract in parallel(24) as described under Appendix B.

A 120 second timeout was applied to both the decompilation procedure and the Soufflé analysis in order to avoid any potential obstructions caused by an extremely complicated contract with a large CFG, or a contract that was unable to be decompiled for a variety of reasons, most likely due to the limitations of the Vandal system. Because we had access to a machine with twenty-four CPUs consisting of two threads each, forty-eight jobs were run in parallel. When one contract was analyzed, the job would move to the next available contract that had not been previously analyzed. The analysis was run in the background using `screen` and detached.

In order to compare the performance of the iterative solution against the integer linear program, a random sample of 25000 contracts was taken out of 886,323. Firstly, the directory containing all contracts was shuffled using the following command:

```
ls runtime-2017-07-28 | sort -R
```

The first 25000 files were then selected and put into another directory, which was analyzed using `python3 analyse.py` that produced a `.dat` file for each edge. Each edge belonging to a particular contract was moved into a separate folder, where a bash file iterated over each directory as described below.

The integer linear programming analysis was conducted using GLPSOL with a model file that contain the integer linear programming solution as described in chapter 3. This model performs the maximum frequency calculations on each edge. When `Python3 analyse.py` was run, each edge for each contract was written into a separate `.dat` file, and was then analyzed with the following command:

```
glpsol --model src/ilp.mod --data <edge.dat>
```

The script was run on each sub-directory concurrently in the background.

The result of each edge was saved into a JSON file in the following format:

```
{from: x, to: y, max_frequency: int, time: z}
```

Where:

- (1) **from**: The inbound vertex
- (2) **to**: The outbound vertex
- (3) **max_frequency**: The maximum frequency count.
- (4) **time**: The time accurate to 10^{-20} seconds.

Each contract consisted of a collection of key-value pairs per edge. The results were stored in three JSON files: one for the iterative maximum frequency calculations, one for the ILP results, and another consisting of the Soufflé analysis. The Soufflé results contained a list of all vulnerabilities each contract was deemed to have (or none at all), in addition to the total runtime of Soufflé for each contract.

Results

The results were generated using a number of Python scripts with libraries including `matplotlib` to produce the numbers below. Firstly, the results of the Soufflé analysis were loaded into Python’s memory using `JSON.load(data)`, producing a list of dictionaries, where each dictionary contains key-value pairs representing the data for that particular contract, such as the decompilation and Soufflé analysis times, and a list containing the names of each exploit detected in the smart contract. The maximum frequency data was also loaded into Python. This data was then analyzed using a variety of custom functions such as calculating the average times below. The gas budget specified for all our analysis was 6.7 million, which is roughly the block gas limit of the network. Table 5.1 describes some basic results.

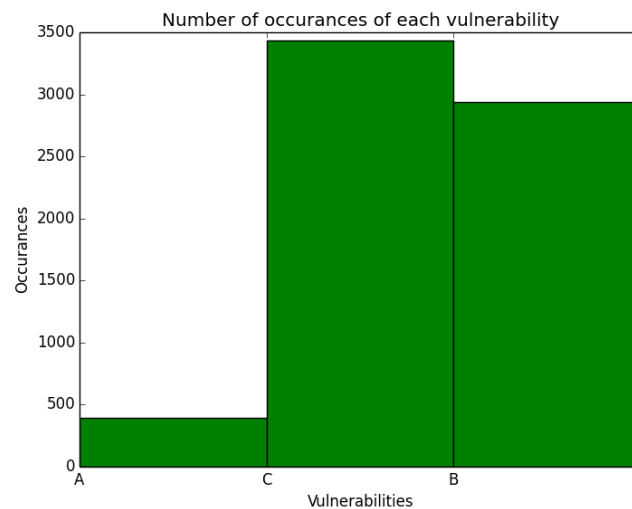


FIGURE 5.1: The number of times a vulnerability has been detected in a smart contract

| Analysis | Number |
|---|-----------------|
| Average Soufflé Runtime Per Contract | 0.28 seconds |
| Average Decompilation Time Per Contract (incl. max frequencies) | 1.08 seconds |
| Total Soufflé Runtime | 250,747 seconds |
| Total Decompilation Time (incl. max frequencies) | 957,444 seconds |
| Total Number of Contracts Analyzed | 519,192 |
| Total Number of Contracts | 886,323 |

TABLE 5.1: Summary of Soufflé and Decompilation Analysis

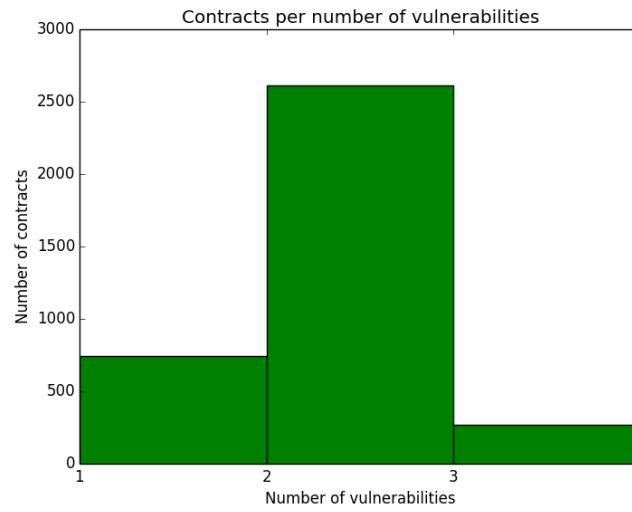


FIGURE 5.2: The number of contracts with a given number of vulnerabilities

Figures 5.1 and 5.2 describe the relationship between the number of contracts and the number of vulnerabilities. Figure 5.1 described the frequency of each vulnerability across each exploit. Each key (A, B and C) corresponds to the following:

- (1) **A:** Wallet Griefing
- (2) **B:** Mass Clearing of Storage
- (3) **C:** Dynamically Bounded Loops

Out of 886,323 contracts, the three vulnerabilities had been detected 394 (0.044%), 3437 (0.388%) and 2935 (0.33%) times respectively. Figure 5.2 describes the number of contracts that have a given number of vulnerabilities, where 746 (0.084%), 2611 (0.295%) and 266 (0.03%) had one, two and three exploits

present respectively. These results are not surprising, given that most contracts in Solidity do not utilize loops. Moreover, the vulnerability detected the most was that of dynamically bounded loops. This is also not surprising, as the presence of the Wallet Griefing vulnerability usually occurs in conjunction of a dynamically bounded loop (for instance, looping over a number of addresses that you wish to send `Wei` to). In addition, all "Mass Clearing of Storage" vulnerabilities are subsets of dynamically bounded loops, as the vulnerability arises out of using `SSTORE` in the context of a loop bounded by size of the mapping or array being cleared, as described in chapter 3.

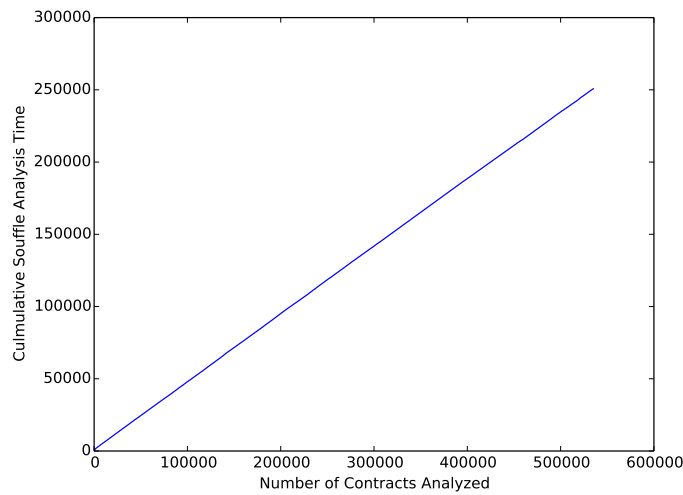


FIGURE 5.3: Cumulative Soufflé analysis time across all analyzable smart contracts

Figure 5.3 describes the cumulative time taken to analyze the total number of contracts that could be analyzed. Note that 519,192 out of 886,323 (58.5%) of all the smart contracts scraped could be successfully analyzed using Soufflé. The other contracts could not be analyzed due to variety of reasons, including that they took too long to decompile and run Soufflé (greater than 120 seconds) or that there are `OP_CODES` or sequence of `OP_CODES` that the decompiler could not understand.

The maximum frequency analysis is divided into two parts. The first contains a list of summarized results as shown in the table below. The second is a comparison between the performance of the integer linear program against the polynomial algorithm.

| Analysis | Number |
|--|----------|
| Average maximum frequencies per contract | 212 |
| Total run time (seconds) | 97847.16 |
| Average run time per contract (seconds) | 0.18846 |
| Percentage of cyclic edges | 1.96% |
| Total contracts analyzed | 519,192 |
| Total Number of Contracts | 886,323 |

TABLE 5.2: Summary of Maximum Frequency Analysis

In the table above, the average maximum frequency number per contracts is 212. The run time is expressed in terms of the iterative algorithm specifically and does not include any other computation outside of that algorithm. The percentage of edges that are cyclic (defined as having a maximum frequency greater than one) is only 1.96%, indicating that cyclic statements are relatively rare in smart contracts, which is not surprising given that loops are generally expensive to run. Note that the number of contracts analyzed with the algorithm is 519,192, exactly the same number that could be analyzed using Soufflé. This is not surprising, given that the creation of the CFG is a necessary conditions to conduct both the maximum frequency and Soufflé analysis.

We also analyze the performance of our iterative solution against our integer linear program. An experiment was conducted on a random sample of 25000 smart contracts from blockchain in order to gauge performance across a wide range of contracts as shown in figure 5.4 and the table below. ILP appears to have performed better, with an average analysis run time per contract of 0.32 seconds against 0.37 for the Iterative solution. This appears counter-intuitive, given that the iterative solution was proven to run in polynomial time as described in Chapter 3, while ILP programs are NP-hard. However, this may be due to a variety of reasons, including the fact that the ILP program might not have run on edges that take too long to analyze, as 11869 contracts were analyzed compared to 13362 for the ILP solution.

Nevertheless, when analyzing several relatively sophisticated contracts, the iterative solution dramatically outperforms ILP. We ran our analysis on four contracts: `owned`, `ownable`, `tokenholder` (all from the Stox ICO token sale contract) (44), and `Token` (9). In total, the average run time was 4.034 and 0.3838 seconds for the ILP and the iterative algorithm respectively, thus showing the expected comparative performance of both. The average number of edges was 61.

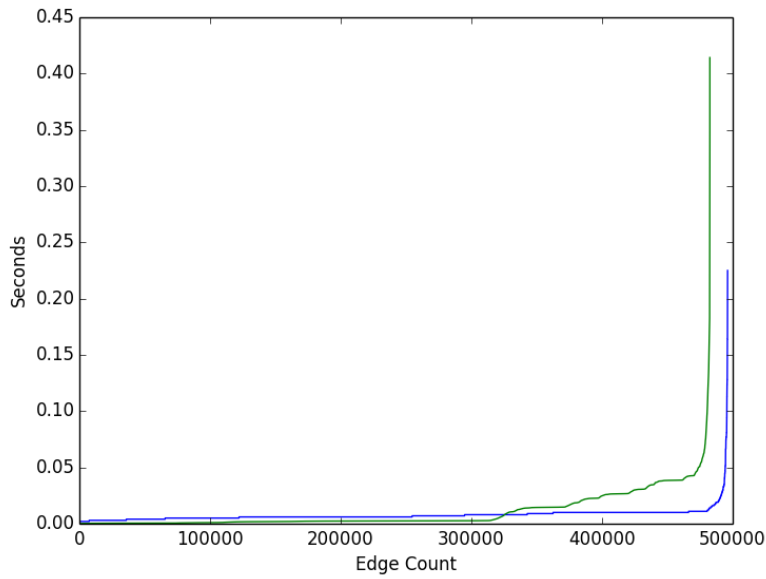


FIGURE 5.4: Performance of ILP vs. iterative solution across a random sample of 25000 contracts. The blue line (bottom) is the performance of the ILP, the green line (top) is the performance of the iterative solution.

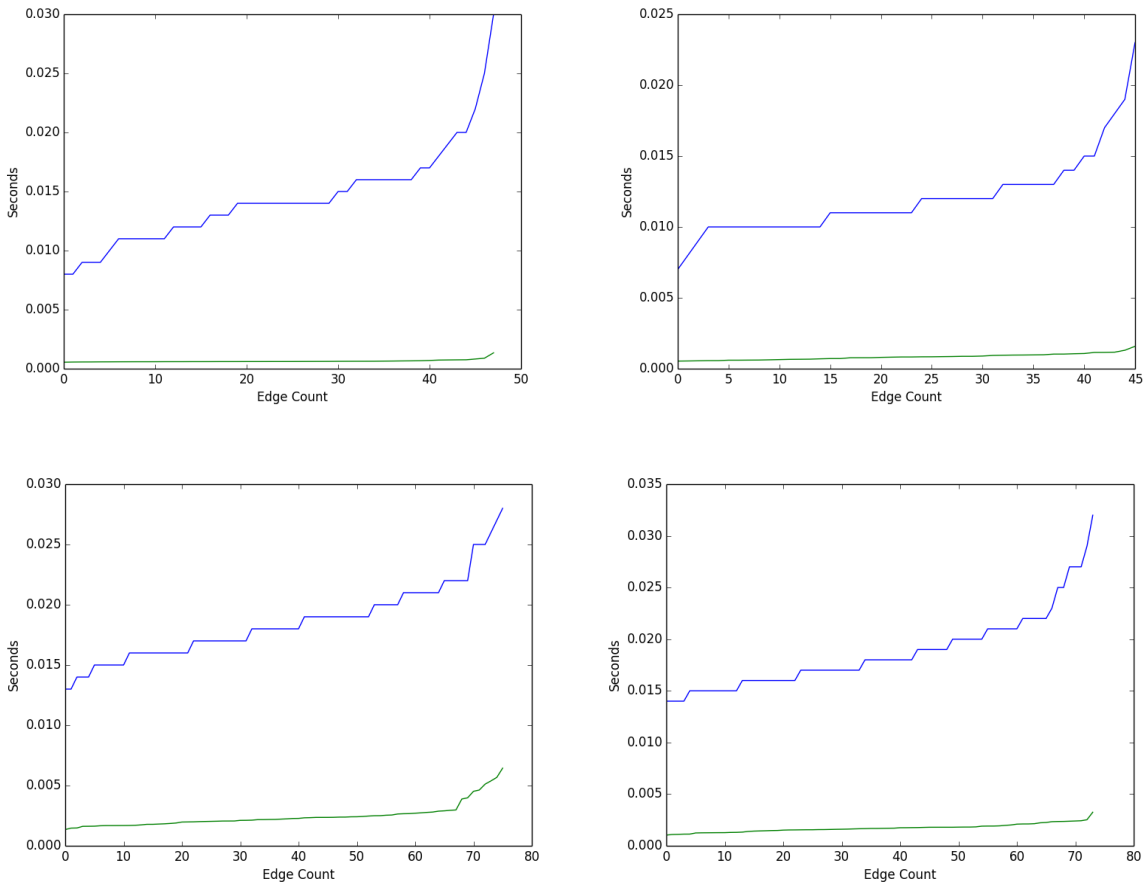


FIGURE 5.5: (Left to right - owned, ownable, tokenholder and Token contracts) ILP vs. Iterative solutions, with time (in seconds) of the y-axis and the number of edges on the x-axis, sorted by time per edge. The blue line (top) is the performance of the ILP, the green line (bottom) is the performance for the iterative solution.

Future Work and Discussion

The work covered in this thesis and in the Vandal Framework can be extended in many ways. Firstly, we can extend our gas analysis and logic specifications to detect gas inefficient coding patterns and hence assist developers in rewriting their smart contracts to be gas efficient. Chen *et al* (2017) developed a gas optimization tool called GASPER. It claims to detect seven gas inefficient patterns. These are

- (1) Unreachable code.
- (2) Opaque predicates.
- (3) Expensive loop operations including the use of `SSTORE`.
- (4) Unnecessary looping, where the result does not change.
- (5) Taking operations outside of a loop.
- (6) Using an unnecessary number of loops when they could be combined together.
- (7) Repeated operations within a loop.

These are all standard programming optimization patterns with significant consequences, as they consume unnecessary gas. Our framework could be extended to detect these patterns, among many others.

In addition, the framework can be extended to detect new vulnerabilities that are being found over time. For instance, the "Ethereum ERC20 short address address attack" allows a malicious user to inject an address that is shorter than 20-bytes long into the data payload sent to a function. For instance, a user should send a data payload consisting of an address and an unsigned integer. A shorter-than 20-byte address would cause the bytes of the unsigned integer (`uint`) to be bit-shifted left, as the EVM adds zeros to the end of the `uint` to create a valid unsigned integer, meaning a malicious user could potentially transfer orders of magnitude more tokens than the smart contract developer intended (47).

Another feature of the EVM that could be analyzed in more depth is `DELEGATECALL` behaviour. `DELEGATECALL` is a message call that allows instructions to be executed in the context of the calling

contract, rather than the Callee unlike `CALL` (19). This significantly increases smart contract functionality. For example, developers can create "upgrade-able" smart contracts by forwarding calls to a contract to another, presumably upgraded contract, while still executing instructions in the context of the original contract (10). However, the behaviour of `DELEGATECALL` can become complex, especially if multiple delegatecalls are made between a chain of smart contracts. This can easily lead to vulnerabilities. For example, a multi-signature smart contract (a smart contract holding Ether and tokens requiring multiple signatures to sign transactions) developed by Parity Technologies and deployed by many organizations was hacked, causing the loss of over 30 million USD worth of Ether. The hack was caused by a `DELEGATECALL` message call whose data was not checked, allowing "all public functions from the library to be callable by anyone" (33). Thus, there is an incentive for the framework to be extended to incorporate the nuance behaviour of `DELEGATECALL`.

Furthermore, automatically detecting underflows and overflows would be incredibly useful. Underflows and overflows occur when a variable (such as `uint`) has an operation performed on it that causes the value to go below zero and hence be set to 2^{256} (the maximum value) or go above 2^{256} and hence become zero (10). This can cause unexpected behaviour. For instance, a malicious user could ask to transfer zero tokens from a function, knowing that the function will perform a deduction on the value without first checking to ensure the result is greater than zero, hence allowing him to transfer out all of the tokens from a smart contract.

Finally, while working on this thesis, Ethereum was upgraded to Byzantium, a major update that represents the first part in the introduction of "Metropolis", the next stage in Ethereum's evolution. Byzantium included nine Ethereum Improvement Proposals (EIPs) which include four new `OP` `CODES`: `REVERT`, `RETURNDATASIZE`, `RETURNDATACOPY` and `STATICCALL` (17). The `require()` assertion available in Solidity is compiled down to `REVERT`, which reverts all changes to state, but unlike an exception returns the remaining call to the caller (5). Both `RETURNDATASIZE` and `RETURNDATACOPY` allows the data to be returned by the EVM (40), while `STATICCALL` is a message call but prevents the callee from modifications to state (5). Incorporating these new `op` `codes` are essential in keeping the Vandal framework up-to-date with Byzantium.

Related Work

Since many bugs appear in smart contract as a result of programming error, several attempts have been made to create other high level programming languages and compilers with a focus on security. Viper is a language with a compiler written in Python3, and claims to have built-in several numeric and type-safely checks that are lacking in the Solidity compiler, such as checking for underflows and overflows, providing a “precise upper bound on the gas consumption of any function call” as well as additional features such as allowing signed integers and fixed point numbers(48). LLL (Low-level Lisp-like Language) is, as the name suggests, a lower level language compared to Viper or Solidity, allowing the developer “direct access to memory and storage” and all EVM Op Codes to optimize and check for vulnerabilities more clearly than analyzing the execution of the EVM itself(12).

Several initiatives and competitions have been conducted in order to discover additional bugs in smart contracts, particularly subtle ones, and to raise awareness in the community of the importance of smart contract security. The underhanded Solidity Coding Contest, taking inspiration from the Underhanded C Coding Contest, is an ongoing contest with multiple rounds. The first round encouraged Solidity programmers to deliberately write small, crowdfunding smart contracts that appear safe, but contain subtle bugs (22). This contest found many serious subtle bugs, such as being able to send Wei to a smart contract without triggering its fallback function (thus potentially leading to unexpected behaviour as the smart contract might not have anticipated it), and the ability to create dynamic arrays of enormous length, leading to underflow and overflow behaviours (the maximum size of an array is 2^{256}) and is being incorporated into future versions of the Solidity compiler (39).

As of V0.4.16, the Solidity Compiler has an in-built SMT Solver with Z3. This allows the developer to test against a variety of checks at compile time that is in-built into the solver, such as underflows and overflows, by importing the checker by activating experimental tools using `Pragma`

`experimental SMTChecker`.(41) The solver also executes assertions and checks the return value of the assertion. For example:

```
assert((a + b) <= 128)
```

will check to ensure that an exception is thrown if `a+b` is less than or equal to 128, which should evaluate to `false`.

There are a number of tools that perform dynamic analysis on behaviour of smart contracts, testing inputs against expected outputs. For instance, “SolCover” is a tool that executes a given set of lines or statements within a particular smart contracts a specified number of times. This form of “code coverage” uses instrumentation by first running “SolParse”, which generates a `.json` file as output containing a hierarchical overview of the contracts and its associated objects, functions and statements and then annotates each statement. These annotations triggers events which are stored in memory for the duration of the analysis (37). SolCover checks for a number of common issues in smart contracts such as checking the return value of `assert` and `require` statements, dead code, and if `if` statements and loops have been written correctly, such as requiring certain vulnerable statements to be inside an `if` statement (i.e a guard statement). However, the tests do not cover a number of factors, such as if a given statement or execution path will lead to an `out-of-gas` exception, as the annotations trigger events (i.e perform additional `LOG` operations) that adds to gas cost. Moreover, SolCover only does analysis on Solidity, **not on the underlying operations of the EVM**. For instance, in earlier versions of SolCover, although `assert(condition)` and `require(condition)` perform exactly the same operations at the EVM as an `if(!(condition)) throw` statement, they were being treated different, with `assert(condition)` and `require(condition)` being treated as branching statements (although this was fixed in later versions of the software).

Several formal verification tools are being developed to identify the exploits described in section one. These tools have been constructed in a variety of different ways which helps illustrate the multiple approaches being used to automatically detect vulnerabilities in improperly written contracts.

Luu *et al* (2016) developed a tool called “Oyente”, a project written in Python which aims to identify a number of vulnerabilities in smart contracts using symbolic execution (29). Their analysis is performed at the EVM level, and is also capable of determining the gas consumption of execution paths.

Oyente identifies four exploits: Transaction-Ordering Dependence, Timestamp Dependence, exceeding the call stack limit of 1024 (Callstack attack) and reentrancy.

However, there are many limitations of their system. Oyente, using symbolic execution, can only analyze an average of nineteen paths per contract, despite taking 350 seconds to run per contract using a significant amount of computer power. In their experiments, Luu *et al* (2016) used 4 Amazon Ec2 10x large instances, 40 CPUS and 160GB of Ram. Hence, many false negatives may exist in a smart contract simply because the execution path with a particular vulnerability was not parsed.

Bhargavan *et al* (2016) provide another formal verification tool for Ethereum Smart contracts. They detect three classes of vulnerabilities. These include checking the return value of external address calls and reentrancy. These patterns were verified in F* by translating the contracts into F* code, from which patterns were written in another F* program and used to detect vulnerabilities. They also converted both bytecode and Solidity into F*, allowing them to perform different analyses on the same contract. By analyzing F* from Solidity, they were able to identify the existence of a reentrant vulnerability. Their software can also compute gas consumption. While translating bytecode into F*, they keep track of each element popped of the stack and add each `OP_CODES` gas costs to a gas counter. This final gas counter is the upper bound for a particular execution path.

Hildenbrandt *et al* (2017) incorporated the entire semantics of the EVM (as of EIP-150) in their framework known as 'K'. Their objective is to create a low-level formal verification tool to identify bugs that could lead to the loss of significant funds. The framework was applied to the "ERC-20" token standard smart contract, which is an interface many developers use when launching tokens to ensure they are compliant with many exchanges. A complete set of instructions was generated from the `TransferFrom(address _from, address _to, uint256 _value)` and `transfer(address _to, uint256 _value)` functions and could check for several properties to ensure that in all cases the functions would not be vulnerable to arithmetic and stack overflows. Their future work involves expanding their analysis to detect a variety of different vulnerabilities, "specifically those that have led to security breaches and financial losses in the past" (26).

Conclusion

This thesis builds upon the existing Vandal framework extending its analysis to loops and resource-based vulnerabilities. The tool can now calculate the total gas consumed for a given execution path in a CFG, calculate upper bounds (maximum frequencies) for the number of iterations for a given loop or edge, and can accurately identify three resource-based vulnerabilities: dynamically bounded loops, Wallet Griefing, and the mass clearing of storage. This is the first framework that can identify and analyze loops, and find vulnerabilities related to resource consumption and loops. These vulnerabilities are easy to create when writing smart contracts, particularly for inexperienced Ethereum developers, and can have serious consequences, especially because once a contract is deployed on the blockchain, it is there forever. Ether or other tokens can be compromised, losing money. For example, the "government", holding 1100 ETH (USD\$330,000) become stuck due to the contract's attempt to clear a large number of elements in storage in a single transaction(15).

In Chapter 2, we outline our approach to building a gas calculator over a CFG, its advantages over other frameworks, and the limitations of such a calculator in the context of static program analysis. We also discussed the three vulnerabilities listed above and provided various examples of how they occur at a high level in the Solidity programming language. In Chapter 3, we briefly described the framework the thesis builds upon, a technical explanation of how these vulnerabilities occur in the EVM and in TAC, and build logical specifications in datalog to identify them. An algorithm for calculating maximum frequencies in polynomial time is constructed and analyzed, as well as a description of our integer linear programming solution. Finally, we outline our results and discuss how other frameworks have attempted to identify vulnerabilities using both dynamic and static program analysis.

Limits of our analysis exist for a variety of reasons. Some operations on the EVM consume a dynamic amount of gas, depending on the data being manipulated. For example, the `SSTORE` operation can consume either 5000 or 20000 gas depending on whether a value is being changed to from a non-zero

value to a zero value or if a non-zero value is being set respectively. Since we take the worst case (20000) for `SSTORE` and all other operations, our framework over-estimates gas consumption. Furthermore, some contracts are very complicated which the framework is unable to decompile and hence we are unable to analyze its associated CFG. Finally, the framework has not been updated to reflect changes in Ethereum's Byzantium updated, which occurred on 16 October 2017. It still reflects the state of the EVM as of Ethereum Improvement Protocol 150's adoption, which occurred over a year ago in response to the spam attack on the Ethereum network (21)

This work can be improved in the future by extending its ability to find new vulnerabilities, identify inefficient coding patterns, incorporate the changes made in Byzantium and future EIPs, and analyze the information flow between multiple smart contracts, which are often inter-twined with one another to create more complex functionality.

Bibliography

- [1] Brent, L. 2017, "escraper2 (EVM Bytecode Scraper 2.0: Distributed Edition)", viewed 28 October 2017, <https://github.com/usyd-blockchain/escraper2>
- [2] Bhargavan *et al* (2016), "Short Paper: Formal Verification of Smart Contracts", *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, 2016*, viewed on 07 November 2017, DOI: <http://dx.doi.org/10.1145/2993600.2993611>
- [3] Buterin, V. 2017, "Thinking About Smart Contract Security", viewed 30 October 2017, <https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security>
- [4] Bylica, P., 2017, "How to Find 10M Just by Reading the Blockchain", viewed on 22 August 2017, <https://blog.golemproject.net/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95>
- [5] Chainskills 2017, "The Byzantium Hardfork Operation", viewed 30 October 2017, <http://chainskills.com/2017/10/10/the-byzantium-hardfork-operation>
- [6] Chen *et al* 2016, "Under-Optimized Smart Contracts Devour Your Money", *SANER(IEEE International Conference on Software Analysis, Evolution, and Reengineering) 2017*, viewed 07 November 2017, doi: 10.1109/SANER.2017.7884650
- [7] Cifuentes, C., Scholz, B., and Zhang, C. 2008, "User-Input Dependence Analysis via Graph Reachability", viewed 30 October 2017, <http://ieeexplore.ieee.org/document/4637536>
- [8] Consensys 2016, "Ethereum, Gas, Fuel Fees", viewed 10 October 2017, <https://media.consensys.net/ethereum-gas-fuel-and-fees-3333e17fe1dc>
- [9] Consensys 2017, "Ethereum Token Contracts", <https://github.com/ConsenSys/Tokens/blob/master/contracts/Token.sol>
- [10] Consensys 2017, "Smart Contract Best Practices", viewed 30 October 2017, https://consensys.github.io/smart-contract-best-practices/software_engineering
- [11] Cooper, K., Harvey, T., and Kennedy, K. (2001), "A Simple, Fast Dominance Algorithm", viewed 24 October 2017, <http://www.hipersoft.rice.edu/grads/publications/dom14.pdf>

- [12] Ellison, 2017, "An Introduction to LLL for Ethereum Smart Contract Development", viewed on 22 August 2017, <https://media.consensys.net/an-introduction-to-lll-for-ethereum-smart-contract-development-e26e38ea6c23s>
- [13] EtherChain. 2017, "Blocks - GetBlocks", viewed on 2 October 2017, <https://etherchain.org/documentation/api/#api-Blocks-GetBlockId>
- [14] EtherChain. 2017, "Verified contract source code available!", viewed on 30 October 2017, <https://etherchain.org/account/0xF45717552f12Ef7cb65e95476F217Ea008167Ae3>
- [15] ethererik, 2017, "GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas", https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck
- [16] Etherscan, 2017, "Etherscan", viewed 28 October 2017, <https://etherscan.io>
- [17] Ethereum Foundation, "EIPs", viewed 30 October 2017, <https://github.com/ethereum/EIPs>
- [18] Ethereum Foundation. 2017, "JSON RPC", viewed on 2 October 2017, <https://github.com/ethereum/wiki/wiki/JSON-RPC>
- [19] Ethereum Foundation. 2017, "Solidity", viewed 10 October 2017, <https://solidity.readthedocs.io/en/develop>
- [20] Ethereum Foundation. 2017, "What is Ether?", viewed 10 October 2017, <https://ethereum.org/ether>
- [21] Jameson, "FAQ: Upcoming Ethereum Hard Fork", <https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork>
- [22] Johnson, N. 2017, "1st Underhanded Solidity Coding Contest", viewed on 2 October 2017, <http://u.solidity.cc>
- [23] Johnson, N. 2017, "How to clear large arrays without blowing the gas limit?", viewed on 17 October 2017, <https://ethereum.stackexchange.com/questions/3373/how-to-clear-large-arrays-without-blowing-the-gas-limit>
- [24] Jurisevic, A. 2017, "Bulk analyser", viewed 28 October 2017, https://github.com/usyd-blockchain/vandal-decompiler/tree/master/tools/bulk_analyser
- [25] Higgins, S. 2017, "Miners Boost Ethereum's Transaction Capacity With Gas Limit Increase", viewed on 2 October 2017, <https://www.coindesk.com/miners-ethereum-transactions-gas-limit>
- [26] Hildenbrandt *et al.* 2017, "KEVM: A Complete Semantics of the Ethereum Virtual Machine", viewed on 7 November 2017, <http://hdl.handle.net/2142/97207>
- [27] McCorry, P., Shahandashti S.F., and Hao, F. 2017, "A Smart Contract for Boardroom Voting with Maximum Voter Pivacy", viewed on 2 October 2017, <https://eprint.iacr.org/2017/110.pdf>

- [28] Luu *et al.* 2017, "global_params.py", viewed on 2 October 2017, https://github.com/melonproject/oyente/blob/4f055b94c37fb56562a4ba2a9dceda3b5e228dbe/oyente/global_params.py
- [29] Luu *et al.* 2017, "Making smart contracts smarter", viewed on 7 November 2017, *ACM CCS 2016*, Doi:10.1145/2976749.2978309
- [30] Nakamoto, S., 2017, "Bitcoin: A Peer-to-Peer Electronic Cash System", viewed on 02 October 2017, <https://bitcoin.org/bitcoin.pdf>
- [31] networkx, "Source code for networkx.algorithms.dominance", viewed 24 October 2017, https://networkx.github.io/documentation/networkx-1.10_modules/networkx/algorithms/dominance.html#immediate_dominators
- [32] Nielson, F., Nielson, H., and Hankin, C. 1999, "Principle of Program Analysis", viewed 24 October 2017, <http://sist.shanghaitech.edu.cn/faculty/songfu/cav/PPA.pdf>
- [33] Palladino, S. 2017, "The Parity Wallet Hack Explained", viewed 30 October 2017, <https://blog.zepplin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7>
- [34] peoplewindow, "Underhanded Solidity Coding Contest". <https://news.ycombinator.com/item?id=14691212>
- [35] Poo Bar, B., 2017, "Assessing The ERC20 Token Exchange Withdrawal Bug / Exploit", viewed on 22 August 2017, <https://www.bokconsulting.com.au/blog/assessing-the-erc20-token-exchange-withdrawal-bug-exploit>
- [36] Rafaloff, E, 2017, "Analyzing the ERC20 Short Address Attack", viewed on 02 October 2017, <https://ericrafaloff.com/analyzing-the-erc20-short-address-attack/>
- [37] Rea, A. 2017, "Code Coverage for Solidity", viewed on 2 October 2017, <https://blog.colony.io/code-coverage-for-solidity-eeefa88668c2>
- [38] Rea *et al.* 2017, "solidity-coverage", viewed on 2 October 2017, <https://github.com/sc-forks/solidity-coverage>
- [39] Reitwiessner, C. 2017, "Lessons Learnt from the Underhanded Solidity Contest", viewed on 2 October 2017, <https://medium.com/@chriseth/lessons-learnt-from-the-underhanded-solidity-contest-8388960e09b1>
- [40] Reitwiessner, C. 2017, "Propose RETURNDATACOPY and RETURNDATASIZE", viewed on 30 October 2017, <https://github.com/ethereum/EIPs/pull/211>
- [41] Reitwiessner, C. 2017, "Version 0.4.16", viewed on 2 October 2017, <https://github.com/ethereum/solidity/releases/tag/v0.4.16>
- [42] Scholz *et al.*, "Datalog", viewed 24 October 2017, <http://souffle-lang.org/docs/datalog>

- [43] Scholz, B. 2017, "Maximum Frequencies", viewed on 2 October 2017.
- [44] Stox, "StoxSmartToken", <https://etherscan.io/address/0x006bea43baa3f7a6f765f14f10a1a1b08334ef45code>
- [45] Suiche, M., 2017, "Porosity: A Decompiler For Blockchain-Based Smart Contracts Bytecode", viewed on 22 August 2017, <https://www.comae.io/reports/dc25-msuiche-Porosity-Decompiling-Ethereum-Smart-Contracts-wp.pdf>
- [46] Vessenes, P., 2016, "Ethereum Griefing Wallets: Send w/Throw Is Dangerous", viewed on 11 June 2017, <http://vessenes.com/ethereum-griefing-wallets-send-w-throw-considered-harmful>
- [47] Vessenes, P, 2017, "The ERC20 Short Address Attack Explained ", viewed on 31 October 2017, <http://vessenes.com/the-erc20-short-address-attack-explained>
- [48] Viper, 2017, "The ERC20 Short Address Attack Explained", viewed on 22 August 2017, <http://vessenes.com/the-erc20-short-address-attack-explained>
- [49] Woods, A. 2016, "DIJKSTRA'S ALGORITHM IN PYTHON", viewed 24 October 2017, <http://alexhwoods.com/dijkstra>
- [50] W, J. 2017, "What are the limits to gas refunds?", viewed on 2 October 2017, <https://ethereum.stackexchange.com/questions/594/what-are-the-limits-to-gas-refunds>
- [51] Wood, G. 2017 "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER EIP-150 REVISION (759dccc - 2017-08-07)", viewed 10 October 2017, <https://ethereum.github.io/yellowpaper/paper.pdf>

APPENDIX A

Four Contracts Analyzed

Below are the raw results of the four contracts analyzed in Chapter 3.

| Stox | owned.hex |
|----------------------------|---------------|
| Number of Edges | 48 |
| Total Run Time (ILP) | 0.69 seconds |
| Total Run Time (Iterative) | 0.031 seconds |

TABLE A.1: Summary of owned.hex ILP vs. Iterative Performance

| Stox | ownable.hex |
|----------------------------|---------------|
| Number of Edges | 46 |
| Total Run Time (ILP) | 0.552 seconds |
| Total Run Time (Iterative) | 0.039 seconds |

TABLE A.2: Summary of ownable.hex ILP vs. Iterative Performance

| Stox | TokenHolder.hex |
|----------------------------|-----------------|
| Number of Edges | 76 |
| Total Run Time (ILP) | 1.407 seconds |
| Total Run Time (Iterative) | 0.187 seconds |

TABLE A.3: Summary of TokenHolder.hex ILP vs. Iterative Performance

| Consensus | Token.hex |
|----------------------------|---------------|
| Number of Edges | 74 |
| Total Run Time (ILP) | 1.385 seconds |
| Total Run Time (Iterative) | 0.128 seconds |

TABLE A.4: Summary of Token.hex ILP vs. Iterative Performance

APPENDIX B

Commands to run analysis

The script “analyse.py” was executed with the following parameters:

```
screen python3 analyse.py -d  
<directory_of_contracts_to_analyze> -j <number_of_concurrent-jobs> -S  
<souffle_executable_binary> -t  
<timelimit_to_analyze_contracts> -T <bailout_limit>
```

For our experimentation, the script was run with the following parameters:

```
python3 analyse.py -d ~/dev/gas-analyzer/scrape-2017-07-28-block4081358  
-j 48 -S ~/dev/souffle/src/souffle -t 120 -T 120
```

To run the decompiler, use the following command:

```
bin/decompile -b <contract_bytecode>.hex
```

This produces a string of output in the terminal consisting of TAC output and the maximum frequency for each edge:


```

---
Entry stack: [V7, 0x1c1, V92]
Stack pops: 0
Stack additions: []
Gas consumed: 1
successor_list: ['0x69f']
block_id: 0x69e

=====

Block 0x69f
[0x69f:0x6a1]
---
Predecessors: [0x69e]
Successors: [0x1c1]
---
0x69f JUMPDEST 1
0x6a0 POP 2
0x6a1 JUMP 8
---
0x69f: JUMPDEST
0x6a1: JUMP 0x1c1
---
Entry stack: [V7, 0x1c1, V92]
Stack pops: 2
Stack additions: []
Gas consumed: 11
successor_list: ['0x1c1']
block_id: 0x69f

=====

Block 0x6a2
[0x6a2:0x6ce]
---
Predecessors: []
Successors: []
---
0x6a2 STOP 0
0x6a3 LOG1 375
0x6a4 PUSH6 0x627a7a723058 3
0x6ab SHA3 30
0x6ad GT 3
0x6af EXP 10
0x6b1 PUSH29 0x29e77893f8a2fd2ff78643cca754cdeaddec1d80bed909ee11b0029 3
---
0x6a2: STOP
0x6a3: LOG S0 S1 S2
0x6a4: V335 = 0x627a7a723058
0x6ab: V336 = SHA3 0x627a7a723058 S3
0x6ad: V337 = GT V336 S4
0x6af: V338 = EXP V337 S5
0x6b1: V339 = 0x29e77893f8a2fd2ff78643cca754cdeaddec1d80bed909ee11b0029
---
Entry stack: []
Stack pops: 0
Stack additions: [0x29e77893f8a2fd2ff78643cca754cdeaddec1d80bed909ee11b0029, V338]
Gas consumed: 424
successor_list: []
block_id: 0x6a2

```

FIGURE B.1: Sample TAC output using the token contract

```
{('1', '0x0'): [54, 0, 0.0015459060668945312], ('0x0', '0x39'): [22, 1, 0.0013620853424072266], ('0x39', '0x44'): [22, 1, 0.0011467933654785156], ('0x44', '0x4f'): [22, 1, 0.001150369644165039], ('0x4f', '0x5a'): [22, 1, 0.001264810562133789], ('0x5a', '0x65'): [7, 1, 0.001135110855102539], ('0x65', '-2'): [0, 1, 0.0011239051818847656], ('0x0', '0x6a'): [19, 1, 0.0011560916900634766], ('0x6a', '0x71'): [6, 1, 0.001154184341430664], ('0x71', '-2'): [0, 1, 0.0011839866638183594], ('0x6a', '0x75'): [109, 1, 0.0012569427490234375], ('0x37b', '0xc9'): [1, 1, 0.0012328624725341797], ('0xc9', '-2'): [0, 1, 0.001191854476928711], ('0x39', '0xcb'): [19, 1, 0.0014772415161132812], ('0xcb', '0xd2'): [6, 1, 0.0014081001281738281], ('0xd2', '-2'): [0, 1, 0.0014967918395996094], ('0xcb', '0xd6'): [15, 1, 0.002006053924560547], ('0x55b', '0xde'): [1, 1, 0.002315998077392578], ('0xde', '-2'): [0, 1, 0.005313873291015625], ('0x44', '0xe0'): [19, 1, 0.002151966094970703], ('0xe0', '0xe7'): [6, 1, 0.001289119720458984], ('0xe7', '-2'): [0, 1, 0.0011420249938964844], ('0xe0', '0xeb'): [15, 1, 0.0012197494506835938], ('0x55d', '0xf3'): [62, 1, 0.0012671947479248047], ('0xf3', '-2'): [0, 1, 0.001207113265991211], ('0x4f', '0x135'): [19, 1, 0.0012412071228027344], ('0x135', '0x13c'): [6, 1, 0.0012087821960449219], ('0x13c', '-2'): [0, 1, 0.0011951923370361328], ('0x135', '0x140'): [15, 1, 0.001219034194946289], ('0x582', '0x148'): [62, 1, 0.0012080669403076172], ('0x148', '-2'): [0, 1, 0.001207113265991211], ('0x5a', '0x18a'): [19, 1, 0.001215219497680664], ('0x18a', '0x191'): [6, 1, 0.0013949871063232422], ('0x191', '-2'): [0, 1, 0.001474142074584961], ('0x18a', '0x195'): [55, 1, 0.0020389556884765625], ('0x69f', '0x1c1'): [1, 1, 0.00164794921875], ('0x1c1', '-2'): [0, 1, 0.0019199848175048828], ('0x75', '0x1c3'): [276, 1, 0.0015308856964111328], ('0x1c3', '0x21b'): [41, 1, 0.0018308162689208984], ('0x21b', '0x23e'): [6, 1, 0.0019562244415283203], ('0x23e', '-2'): [0, 1, 0.001943111419677344], ('0x21b', '0x242'): [41, 1, 0.0019478797912597656], ('0x242', '0x265'): [6, 1, 0.0015716552734375], ('0x265', '-2'): [0, 1, 0.0017168521881103516], ('0x242', '0x269'): [46, 1, 0.0018031597137451172], ('0x269', '0x2a1'): [6, 1, 0.0013952255249023438], ('0x2a1', '-2'): [0, 1, 0.00131988525390625], ('0x269', '0x2a5'): [875, 1, 0.0012009143829345703], ('0x2a5', '0x34c'): [6, 1, 0.001180887222290039], ('0x34c', '-2'): [0, 1, 0.0011701583862304688], ('0x2a5', '0x350'): [728, 1, 0.0011951923370361328], ('0x350', '0x35d'): [6, 1, 0.0012288093566894531], ('0x35d', '-2'): [0, 1, 0.0012462139129638672], ('0x350', '0x361'): [43, 1, 0.001237630844116211], ('0x361', '0x373'): [1, 1, 0.001232147216796875], ('0x373', '0x374'): [1, 1, 0.001222848892211914], ('0x374', '0x375'): [3, 1, 0.0012290477752685547], ('0x375', '0x377'): [3, 1, 0.0011897087097167969], ('0x377', '0x379'): [3, 1, 0.0011892318725585938], ('0x379', '0x37b'): [15, 1, 0.0011861324310302734], ('0xd6', '0x380'): [276, 1, 0.0011327266693115234], ('0x380', '0x3d8'): [6, 1, 0.0011200904846191406], ('0x3d8', '-2'): [0, 1, 0.0013020038604736328], ('0x380', '0x3dc'): [41717, 1, 0.0011408329010009766], ('0x3dc', '0x55b'): [9, 1, 0.0011372566223144531], ('0xeb', '0x55d'): [251, 1, 0.0013709068298339844], ('0x140', '0x582'): [251, 1, 0.0014369487762451172], ('0x195', '0x5a8'): [276, 1, 0.001353979110717734], ('0x5a8', '0x600'): [280, 1, 0.0011281967163085938], ('0x600', '0x658'): [6, 1, 0.0011138916015625], ('0x658', '-2'): [0, 1, 0.0014519691467285156], ('0x600', '0x65c'): [20268, 1, 0.002313852310180664], ('0x65c', '0x69e'): [1, 1, 0.001967906951904297], ('0x69e', '0x69f'): [11, 1, 0.0019562244415283203], ('1', '0x6a2'): [424, 0, 0.0018548965454101562], ('0x6a2', '-2'): [0, 1, 0.0013570785522460938]}
```

FIGURE B.2: Sample maximum frequency output using the token contract

APPENDIX C

Screenshot of Python maximum frequency scripts

```
cfg = self.source
tuple_list = []
# Convert basic blocks to edges
# add start node
# Start node: -1
# End node: -2
for n in cfg.blocks:
    # If no predecessors, the start node becomes the predecessor
    if len(n.predecessors) == 0:
        tuple_list.append(['-1', n.block_id, n.gas_consumed])
    else:
        # For each predecessor, generate a new edge (tuple) where predecessor is the initial outgoing vertex
        for p in n.predecessors:
            tuple_list.append([p, n.block_id, n.gas_consumed])
    # If no successors, then tuple has no outgoing edges. Create outgoing edge to end node.
    if len(n.successors) == 0:
        tuple_list.append([n.block_id, '-2', 0])
```

FIGURE C.1: Screenshot of code that creates a common start and end node, and converts basic blocks into weighted edges

```

def create_graph(self, basic_block_list, gas_budget):
    """
    Creates a Directed Weighted Graph
    """
    # Create an empty directed weighted graph
    g = dijkstras.Graph()
    list_of_verticies = []
    # Time take to execute max freq calculation on each edge pair.
    time_taken = 0
    # Construct graph by adding nodes and edges.
    for index in enumerate(basic_block_list):
        g.add_vertex(str(index[1][0]))
        g.add_vertex(str(index[1][1]))
        g.add_edge(str(index[1][0]), str(index[1][1]), index[1][2])
    for k, v in g.weights.items():
        # Get current time
        start_millis = time.time()
        sp_s_p = dijkstras.shortest_path_with_weights(g, str(basic_block_list[0][0]), k[0])
        sp_q_e = dijkstras.shortest_path_with_weights(g, k[1], str(basic_block_list[len(basic_block_list) - 1][1]))
        # Check if cycle exists
        cycle = dijkstras.shortest_path(g, k[1], k[0])
        # If no path from s to p, p to q, q to e, set max freq to 0
        if len(sp_s_p) < 3 or len(sp_q_e) < 2:
            v.append(0)
        # Does the inequality  $sp(s,p) + w(p,q) + sp(q,e) \leq B$  hold:
        elif (sp_s_p[len(sp_s_p) - 1] + v[0] + sp_q_e[len(sp_q_e) - 1]) <= gas_budget:
            if len(cycle) == 1:
                # No cycle, set max freq = 1
                v.append(1)
            elif len(cycle) > 1:
                # Cycle exists of some length
                v.append(1 + math.floor((gas_budget - sp_s_p[len(sp_s_p) - 1] - v[0] - sp_q_e[len(sp_q_e) - 1]) / (v[0] + sp_q_e[len(sp_q_e) - 1])))
        else:
            v.append(0)
        end_millis = time.time()
        # calculate the time taken to run max freq calculation for a given edge
        time_taken = end_millis - start_millis
        # add that to list
        v.append(time_taken)
    return g.weights

```

FIGURE C.2: Screenshot of the function that returns the maximum frequencies per node using Dijkstras' algorithm