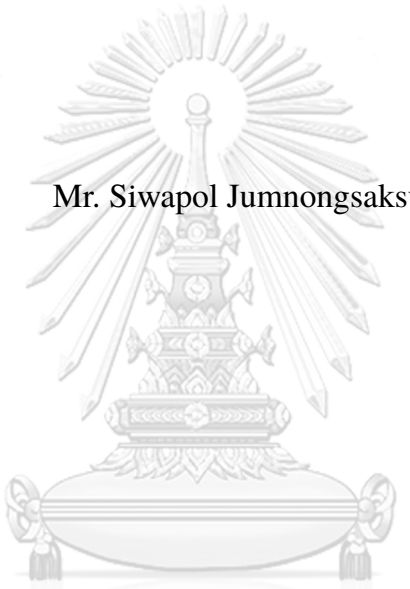
การลดความผิดพลาดที่เกิดขึ้นในขณะทำงานของสัญญาอัจฉริยะบนอีเทอเรียมบล็อกเชน

นายศิวะพล จำนงศักดิ์ทรัพย์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2563

REDUCING SMART CONTRACT RUNTIME ERRORS ON THE ETHEREUM BLOCKCHAIN

Mr. Siwapol Jumnongsaksub



A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2020

Thesis Title	REDUCING SMART CONTRACT RUNTIME ERRORS ON THE ETHEREUM BLOCKCHAIN
By	Mr. Siwapol Jumnongsaksub
Field of Study	Computer Engineering
Thesis Advisor	DR. Kunwadee Sripanidkulchai, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

. . . . . Dean of the Faculty of Engineering
(Professor Supot Teachavorasinskun, D.Eng.)

THESIS COMMITTEE

. . . . . Chairman
(Associate Professor Kultida Rojviboonchai, Ph.D.)

. . . . . Thesis Advisor
(DR. Kunwadee Sripanidkulchai, Ph.D.)

. . . . . Examiner
(DR. Duangdao Wichadakul, Ph.D.)

. . . . . External Examiner
(Assistant Professor Sukumal Kitisin, Ph.D.)

ศิวะพล จำนงศักดิ์ทรัพย์: การลดความผิดพลาดที่เกิดขึ้นในขณะทำงานของสัญญาอัจฉริยะบนอีเทอเรียมบล็อกเชน. (REDUCING SMART CONTRACT RUN-TIME ERRORS ON THE ETHEREUM BLOCKCHAIN) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : อ. ดร. กุลวดี ศรีพานิชกุลชัย, 84 หน้า.

สัญญาอัจฉริยะทำให้แอปพลิเคชันหลากหลายประเภทสามารถทำงานบนบล็อกเชนได้ อีเทอเรียมได้ทำการเก็บรหัสไบต์ของสัญญาอัจฉริยะไว้กับที่อยู่ของสัญญาอัจฉริยะ เพื่อให้อีวีเอ็มสามารถอ่านและทำธุรกรรมได้อย่างถูกต้อง ธุรกรรมที่ได้ดำเนินการไปแล้วทั้งหมดไม่ว่าจะสำเร็จ หรือ ล้มเหลว จะถูกจัดเก็บอยู่ในฐานงาน โดยไม่สามารถแก้ไขได้ตลอดไป ธุรกรรมที่ล้มเหลวเหล่านี้ถูกหยุดการทำงานโดย อีวีเอ็ม เนื่องจากข้อผิดพลาดขณะดำเนินงาน และ ส่งผลให้เกิดการสิ้นเปลือง โดยมีมูลค่าที่สูญเสียที่ผ่านมาแล้วอย่างน้อย 2 ล้านอีเทอร์ หรือ คิดเป็น 634.2 ล้านดอลลาร์สหรัฐ วิทยานิพนธ์นี้ได้เสนอ Evitar อัลกอริทึมสำหรับแจ้งเตือน เพื่อลดข้อผิดพลาดขณะดำเนินงานของสัญญาอัจฉริยะของอีเทอเรียม โดยมีวิธีการสองวิธี วิธีการแรก Evitar แนะนำให้ผู้ใช้งานส่งธุรกรรมด้วยจำนวนแก๊สสูงสุด เพื่อหลีกเลี่ยงการเกิดข้อผิดพลาดที่เกิดจากแก๊สไม่เพียงพอ อย่างไรก็ตาม เมื่อธุรกรรมล้มเหลวจะก่อให้เกิดค่าธรรมเนียมในการทำธุรกรรมที่สูง เพื่อป้องกันปัญหานี้ วิธีการที่สอง Evitar ได้ทำการวิเคราะห์ธุรกรรมที่เรียกไปยังชุดคำสั่งต่าง ๆ ในสัญญาอัจฉริยะ และ ระบุว่าเป็นชุดคำสั่งที่มีโอกาสล้มเหลวสูง ในกรณีที่จำนวนธุรกรรมที่ล้มเหลวมีจำนวนถึงจุดที่กำหนดไว้ วิธีการนี้จะป้องกันไม่ให้ผู้ใช้งานส่งและจ่ายค่าธรรมเนียมสำหรับธุรกรรมที่มีแนวโน้มว่าจะล้มเหลว วิทยานิพนธ์นี้ได้ทำการทดลองเพื่อวัดประสิทธิภาพของ Evitar โดยทำการส่งธุรกรรมทั้งหมดใหม่อีกครั้งในระบบโครงข่ายส่วนตัว ผลการทดลองแสดงให้เห็นว่า Evitar สามารถลดธุรกรรมที่ล้มเหลวได้ถึง 99.52 เปอร์เซ็นต์ เมื่อเทียบกับการส่งธุรกรรมแบบปกติ โดยพบการลดลงของธุรกรรมที่สำเร็จ 1.78 เปอร์เซ็นต์ ในขณะที่ปริมาณแก๊สที่ใช้โดย Evitar น้อยกว่าการส่งธุรกรรมแบบปกติมากถึง 10 เท่า สำหรับการส่งธุรกรรมด้วยจำนวนแก๊สสูงสุดสามารถลดการเกิดข้อผิดพลาดที่เกิดจากแก๊สไม่เพียงพอได้ถึง 99.25 เปอร์เซ็นต์ นอกจากนี้ Evitar ยังสามารถประหยัดพื้นที่จัดเก็บเป็นจำนวน 15.04 กิกะไบต์ หรือ ประมาณ 82.32% ในโหนดปกติ และ 17.04 กิกะไบต์ หรือ ประมาณ 50.09% ในโหนดสำหรับเก็บข้อมูลถาวร

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	.....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาหลัก	.....
ปีการศึกษา	2563		

6270336821: MAJOR COMPUTER ENGINEERING
KEYWORDS: BLOCKCHAIN, SMART CONTRACT, RUNTIME ERROR

SIWAPOL JUMNONGSAKSUB : REDUCING SMART CONTRACT RUNTIME ERRORS ON THE ETHEREUM BLOCKCHAIN. ADVISOR : DR. Kunwadee Sripanidkulchai, Ph.D., 84 pp.

With smart contracts, a wide range of applications can be implemented on blockchains. Ethereum stores smart contract byte code with the smart contract address so, the Ethereum Virtual Machine (EVM) can read and execute transactions correctly. All executed transactions (both successful and failed transactions) are stored on the platform permanently. Failed transactions are thrown by the EVM due to runtime errors and result in monetary waste. The waste from these transactions add up to around 2 million Ethers or $634.2 million. In this thesis, we propose Evitar, a warning algorithm for reducing Ethereum smart contract runtime errors, which has two mechanisms. First, Evitar proposes that users send transactions with the maximum gas allowed to avoid Out of Gas errors. However, this results in an extremely high transaction fee when transactions fail. Second, Evitar analyzes transactions called to each method in smart contracts and marks a method as a method with a high failure rate if the number of failed transactions reaches Evitar's threshold. This mechanism prevents users from sending and paying for transactions that are likely to fail. We run experiments to evaluate the performance of Evitar by replaying transactions in a private network. The results show that Evitar can reduce failed transactions up to 99.52% compared to sending under default behaviour in exchange for a reduction in successful transactions by 1.78%. The amount of gas used by Evitar is only one-tenth compared to sending under default behaviour. Sending transactions with the maximum gas in Evitar reduces Out of Gas errors by 99.25%. In addition, Evitar can save up to 15.04 GB (82.32%) of storage in the Geth default

node and 50.09 GB (50.09%) in the Parity full archive node.

Department:	Computer Engineering	Student's Signature	.....
Field of Study:	Computer Engineering	Advisor's Signature	.....
Academic Year:	2020		

Acknowledgements

I would like to thank Dr. Kunwadee Sripanidkulchai, my advisor, for giving me an opportunity to learn about blockchain especially Ethereum blockchain and inviting me to enrol in graduate school. She freely let me choose my thesis topic and always give me a lot of advice to make my topic more feasible. When I struck with any problems during my research, she always discusses with me for solutions. For the time I feel tired and burnout, she cheers me up and gives me motivations to continue on the research. Talking with her always broaden my options not only about research but it is including student life, teaching skill, and how to think carefully before deciding.

I thanked thesis committees including Associate Professor Kultida Rojviboonchai, DR. Duangdao Wichadakul, and Associate Professor Sukumal Kitisin for giving valuable advice to improve this thesis.

I thanked the scholarship from the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.

I thanked my beloved family who was also another person to give me an opportunity to get into higher education and always listen to every problem during my research. They also gave me morale when I feel tired during my research.

I also thanked my ISEL lab mates for giving valuable advice and helping me to solve research problems during these two years.

CONTENTS

Page

Page

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

Chapter I

INTRODUCTION

Blockchains are disrupting computing platforms in many sectors, such as the financial, supply chain, and healthcare industries. One of the most practical use cases of blockchains is cryptocurrency [4, 5]. Cryptocurrency is a digital medium of exchange in which all transaction records are stored in a blockchain. Bitcoin [6], the first well-known cryptocurrency with the largest market cap [7], released in January 2009 by Satoshi Nakamoto, gives the potential to perform a financial transaction online without a trusted bank. Anyone can send and receive Bitcoin by participating in the Bitcoin network and sending Bitcoin via transactions. Another use case of blockchains is supply-chain. Using blockchains in the supply-chain industry, customers gain the traceability of products including material, process, transportation, and even certification [8]. All records of the product are stored publicly in the blockchain and can be accessed anytime. For the healthcare sector, blockchains are used to store the right to access patient personal information such as past medical history, family history, and social history [9].

At its core, a blockchain platform implements an immutable distributed ledger whose content is propagated to all nodes in a peer-to-peer network and treated as global state. With a distributed ledger concept, all nodes have to eventually hold the same data. This makes all data stored on blockchains transparent to the public. A node that holds a different data from others is marked as a malicious node and is rejected from other nodes in the network. To participate in a network, a user needs to set up a node to run the blockchain client and connect to other nodes in the network as peers to sync up its database. When its database is synced, a node is available and able to make transactions. Since blockchains are transparent, all blockchain state including current and previous state can be accessed at any time. In order to update state in the blockchain, a user must create a transaction with the

updated data then sign the transaction with the signature. After a transaction is signed, the user needs to relay the transaction to its peers. Validators (miners) then pick transactions for validating and bundle them with other validated transactions. Validators are users that provide their resources to validate the correctness of transactions. All validated transactions are stored in a block along with data from the latest block to form a chain-like database. Finally, the validator publishes the block to the network. If the block is accepted by the network, it is stored in the blockchain and the validator receives a reward for the validation. With the immutable properties and mechanisms to validate transactions, blockchain technology can overcome the double-spending problem [10, 11] which is one of the most critical challenge in the financial sector. The double-spending problem is the risk that malicious users can spend digital currency twice due to the latency of the network as some nodes may not have the updated data. With blockchain technology, the network can be formed and run without trusting other nodes while preventing a malicious node with a counterfeit dataset to participate.

Ethereum [12] is one of the most popular blockchains with the second-largest market-cap [7]. Ethereum is integrated with the Ethereum Virtual Machine (EVM) [12]. Ethereum can compile and execute smart contracts [13, 14]. A smart contract is a set of code that is deployed in the Blockchain and can be executed when triggered by related transactions. With the EVM and smart contracts, Ethereum has the potential to run many applications such as product traceability [15, 16, 17], e-voting [18, 19], and games [20, 21, 22]. In addition, smart contracts can enable new decentralized financial systems such as lending platforms [23] for supplying or borrowing cryptocurrency with auto-calculated interest and decentralized exchanges [24].This gives users an opportunity to provide liquidity for each trading pair while receiving some of the trading fees as a reward. To use a smart contract on a blockchain, a user writes smart contract code and deploys it to the Ethereum network. Once a smart contract is mined, other users can trigger methods in the smart contract by sending transactions to the smart contract with proper input.

While smart contracts have been widely deployed on the Ethereum platform, not all smart contract transactions are successful. Some transactions fail due to runtime errors thrown by EVM during executions. When the EVM detects a fatal error, it halts current execution, reverts global state, and marks the transaction as failed. These failed transactions cost significant waste in terms of resources for execution, storage of transaction data, and money for transaction fees and block rewards. We collect and analyze transaction data from the Ethereum public network which is the primary public network of the Ethereum blockchain where transactions are sent and mined. The analysis shows that there are around 2 million Ethers wasted from these transactions which can be divided into 30,430 Ethers for transaction fees and 1.96 million Ethers for block rewards from generating blocks to store these transactions. In order to estimate the waste in term of fiat currency, these wasted Ethers are converted to dollars using the daily exchange rate. For the transaction fee, 30 thousand Ethers is worth $13 million while the 1.96 million Ethers for block rewards is worth $621.2 million. This means that a huge amount of money is generated and used wastefully. This existing waste cannot be reduced due to the immutable property of blockchain but new incoming potential waste can be reduced if users can avoid sending transactions that are likely to result in failure.

Our work attempts to quantify runtime errors, provide an algorithm to detect smart contract methods with bad transaction behavior, and give out warning messages to prevent further transactions on methods that are likely to fail. We analyze the behavior of runtime errors from transactions. We find that some errors are necessary because the required logic and network state may not have been known prior to execution. However, many of them can be avoided as they are caused by human errors: poorly written contracts and poorly funded transactions. In this thesis, we design Evitar [25], a dynamic algorithm that analyzes smart contract-related transactions to identify each method's transaction behavior. We label any methods as method with a high failure rate if many of the transactions that called the method failed. Evitar's main purpose is to reduce the number of failed transactions on the Ethereum blockchain by helping users to avoid sending transactions that will result

in failure. This will improve the transaction processing performance of the overall network since transactions that are likely to fail are prevented and most of the transactions selected by miners for validation are successful transactions. Furthermore, we can reduce the overall storage of the blockchain by storing fewer failed transactions.

Chapter II

BACKGROUND

The related technologies to reduce failed transactions in the Ethereum network are reviewed: First we discuss blockchains, a peer-to-peer network with a distributed database. Next we discuss Ethereum, an open-source blockchain that provides a platform for transferring cryptocurrency and running smart contracts. In addition, Ethereum is integrated with the Ethereum Virtual Machine (EVM) which gives Ethereum the potential to execute smart contracts, on its platform. Last, we describe all types of EVM runtime errors.

2.1 Blockchain

A blockchain is a digital ledger where all records are duplicated and stored on all nodes in a peer-to-peer network. Blockchain is an example of distributed ledger technology (DLT) [26].All stored records are immutable. New records are updated on all nodes in the same order. This means that anyone who updates a different set of data from the majority is marked as a malicious user and loses trust in the network. Data recorded in the blockchain is persistent and data modification is not permitted. A blockchain has two main structural parts which are the block and the chain as shown in Figure 2.1. In a blockchain data structure, blocks are linked one-by-one to form a chain-like database which means that there is only one chain that is valid. If the chain has diverged, the longer chain will be chosen and the shorter one is discarded. Each block has a header and body. The block's header contains the previous block's hash which is the key field for linking blocks together as well as a timestamp, block number, nonce, and Merkle root [27]. The body contains a set of transactions that are validated transaction. Transactions can be created by anyone in the network and sent to the transaction pool to wait for validation. In a public blockchain, anyone can participate as a miner to validate the correctness

of transactions, add them to a block, and publish blocks to the network. Once a published block is accepted by the others in the network, the miner receives a reward.

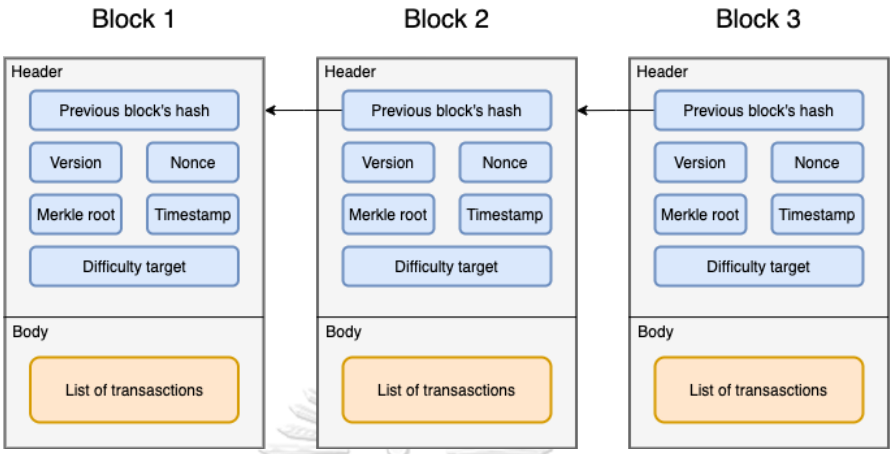


Figure 2.1: Blockchain's block structure.

2.2 Ethereum

Ethereum is one of the most well-known blockchains. Ethereum is the first blockchain that enables smart contracts integration through the Ethereum Virtual Machine (EVM). With the EVM and smart contracts, Ethereum transactions go beyond sending and receiving assets. Ethereum users can send transactions to create smart contracts and call them. Although smart contracts enable more use cases on Ethereum, they increase the demand for throughput on the Ethereum network since smart contracts are called via transactions. Furthermore, some transactions can result in failure due to insufficient gas or unsatisfied conditions during execution. Even if a transaction results in failure and does not affect any state in the blockchain, it still is stored in the blockchain permanently.

Ethereum is based on the the DLT concept like Bitcoin which means that the Ethereum network also forms a single chain database. Ethereum uses an account-based model [13] in which users hold a private key to access their account. The account holds the user is current balance and is used to exchange Ethers with other accounts. To support smart contracts, Ethereum adds a new type of account for smart contracts which can store smart contract bytecode and can be automatically

executed when there are transactions calling them. Ethereum also uses a gas model as an incentive for miners. The gas model is used to compute the transaction fee that owners of transactions give to miners for transaction execution. There are two factors for calculating transaction fees. The first factor is the amount of gas provided. Ethereum transactions cost 21,000 gas plus 4 and 68 for each zero and non-zero byte of input respectively. If the transaction is for a smart contract, gas cost is higher because more operations are performed by the EVM. Gas cost for each operation in the EVM are as shown in Appendix A. Second is the gas price, which is the price of each unit of gas spent for the execution. Mostly, miners prioritize transactions with a higher gas price. Transaction fees that miners receive are calculated by multiplying gas price with the amount of gas used for the execution.

To update the protocol on the Ethereum blockchain, all nodes need to update their client to the latest one. There are two types of protocol updates: soft fork (SF) and hard fork (HF). The SF is a minor update so nodes running older protocol versions are still compatible. However the HF is a chain split. Only updated nodes can participate while nodes with older protocol version need to update before participating. In every HF, the block number to activate a new update is specified so that nodes in the network know when to update. There are many important HFs [28] in Ethereum such as the Homestead HF [29], Byzantium HF [30], Constantinople HF [31, 32], and Istanbul HF [33].

2.3 Ethereum Virtual Machine (EVM)

The Ethereum Virtual Machine or EVM is a virtual machine that is integrated with the Ethereum client for handling smart contract deployment and execution on the Ethereum network. It is a quasi-Turing-complete state machine with limited execution processes which are controlled by the amount of gas provided by a sender. The EVM is built as a virtual stack machine with a maximum stack size of 1,024 stacks and each stack item has a size of 256 bits. The EVM also contains a predefined instruction set called opcodes [12, 34],where each opcode is an 8-bit unsigned

integer. The EVM is used when a transaction for creating or calling smart contract is selected for validation. It starts by loading the smart contract current state and transaction input, then executes the transaction following the opcodes read from transaction input, and uses user-supplied gas as fuel. During the execution, gas serves as an upper-bound on the amount of work performed by the EVM to prevent the execution from endless work. If the EVM detects a fatal error during execution, the EVM handles it by halting the current execution. After the EVM halts the execution, it reverts global state back to the initial state prior to the transaction and marks the transaction as failed.

2.4 Smart Contracts

A smart contract on Ethereum is a set of agreements in the form of code which can be self-executed when a transaction triggers it. An Ethereum smart contract is written in higher-level languages such as Solidity [35] and Vyper [36], then compiled into EVM bytecode. Once a smart contract is created it can never be updated. However, it can be deleted using a self-destruct opcode [12, 34]. Figure 2.2 shows the smart contract workflow. First, the smart contract owner writes the smart contract source code in an editor and compiles the source code into EVM bytecode. Then, the owner deploys the smart contract bytecode into the Ethereum network by sending a transaction with the compiled input and waits for the transaction to be mined. After the transaction is mined, others can call the smart contract specifying the method hash and variable as inputs to execute. Smart contract transactions can be divided into two types: smart contract *created* transactions and smart contract *called* transactions which have different input formats and receivers. For the former, there is no receiver since it is used for creating a new smart contract. The input format consists of three parts: creation code which is used by the EVM to initialize state for the smart contract when a smart contract is created, runtime code which is the main part that contains all code for smart contract execution when called, and swarm code which is hash of smart contract metadata and can be used to query

a decentralized storage system called Swarm. For called transactions, the first 10 characters (including "0x") is the method hash which the transaction needs to use to trigger execution. A method hash is generated by hashing the method's name, arguments and argument's types. Method hashes that are the same may have different code inside. The rest of the input is arguments of the method represented as hex data. The called transaction type without a receiver is a smart contract transaction, but labeled as failed due to its invalid input format (i.e., missing a receiver).

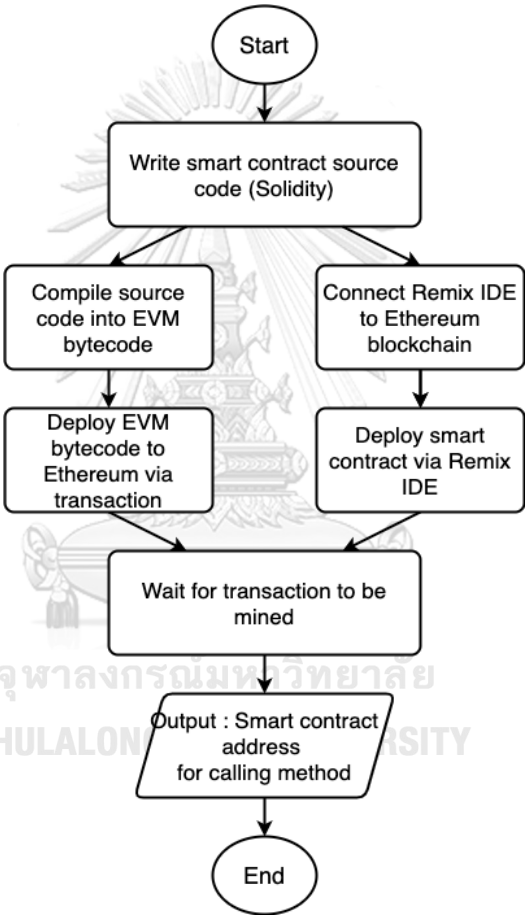


Figure 2.2: Flowchart of smart contract creation.

2.5 EVM Runtime Error

From the Ethereum documentation, the EVM handles failure situations by throwing the current execution, reverting the current state back to the initial state, handling transaction gas, and marking that transaction as failed. This means that

failed transactions will not change any global state but still consume gas as fuel for execution (i.e., reward for the miner). These failed transactions are stored in the blockchain, the same way as successful transactions. The EVM has 7 types of runtime errors [12, 30] with different gas consumption behaviors described below.

1. Out of Gas : This runtime error occurs when the gas provided by the sender for the smart contract execution is insufficient. In depth, the cause of this runtime error can come from poorly written smart contract source code such as superfluous code, multipurpose methods, or endless loops. An Out of Gas error consumes all gas provided by the user because it is all used up.

2. Revert : This is a require-style exception thrown by the revert opcode which was introduced in the Byzantium HF. Reverts are typically used by smart contract creators for validating smart contract inputs or conditions inside methods to ensure programmatic correctness, such as checking contract ownership, account balance, or current time. The gas consumption of a Revert error is different from other runtime errors. It only partially consumes gas used from the beginning if execution to the point that the revert is detected.

3. Invalid Opcode : This is an assert-style exception thrown by invalid opcodes usually to prevent mathematical errors such as divide by zero. Generally, this error is unacceptable and should not occur as it should have been caught by condition-checking code in the contract. An Invalid Opcode consumes all gas provided as a penalty since these runtime error must not occur.

4. Invalid Jump : This is an old error that has been replaced by revert. However, it can still occur with older published contracts (Solidity versions prior to 0.4.10). The cause of this runtime error is when the EVM calls the jump opcode and the jump destination is invalid or non-existent. This runtime error consumes all gas, same as the Invalid Opcode error. This was considered to be too much gas consumed and was later replaced by the Revert error.

5. Stack Underflow : This runtime error happens when the EVM pushes undefined opcodes to the stack or pops an empty stack. This is caused by malformed inputs usually during smart contract creation with two common patterns. First, the user created the smart contract transaction with incomplete inputs. So, the EVM executes that transaction accidentally pushing undefined opcodes to the stack. Second, the user called a smart contract transaction without specifying a receiver. This makes the EVM interpret it as a smart contract creation with malformed input instead. This runtime error also consumes all gas provided.

6. Stack Overflow : This runtime error happens when the EVM pushes an opcode to a full stack. The EVM is a stack machine with a limit of 1024 stack size. During execution, the call opcode [34] is usually pushed on the stack. The call has to be executed completely before it can be popped out. Recursively calling this opcode will cause the EVM to keep storing call opcodes in its stack, exceeding the stack size causing Stack Overflow. This runtime error also consumes all gas provided.

7. Mutable Call in Static Context : This runtime error happens when the EVM tries to call the *STATICCALL* opcode with the modification in the context that stores global state. Staticcall is used to call read-only functions. When staticcall is used for any function that modifies state, the EVM halts the execution and returns this runtime error.

The execution status, successful or failed, can be found in the transaction receipt after the Byzantium block. However, the runtime error which is the root cause of the transaction, is not stored directly with the transaction or its receipt. The runtime error of transactions can be obtained using the transaction trace which will be further explained in Section 4.1.

Chapter III

RELATED RESEARCH

The related work focuses on smart contract vulnerabilities and security issues by analyzing smart contract source code and bytecode. Solutions and tools are proposed for helping smart contract creators to write code with less vulnerabilities and avoid using deployed vulnerable smart contract.

3.1 MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts [1]

In the Ethereum blockchain, gas is the fuel for executing smart contracts. Users must pay gas upfront so that the execution cost may not exceed the provided gas. Some transactions fail due to insufficient gas provided while some of them fail due to vulnerabilities in smart contracts. Smart contracts that does not handle the possible abortion of transactions may contain gas-focused vulnerabilities. A vulnerable smart contract must not be called because calling it will always end up with insufficient gas. To prevent this, developers and auditors need to make extensive use of programming techniques and many tools to minimize the risk of being attacked by attackers. Smart contracts that do not handle the possible abortion of a transaction correctly are at risk for a gas-focused vulnerability. The examples of gas-focused vulnerabilities are described as following.

1. Unbounded Mass Operations : This vulnerability is found in smart contracts that contain too many loops. When executed, it consumes too much gas or loops infinitely. To close this vulnerability, loop conditions in smart contracts must be validated with the maximum gas allowed for execution.

2. Wallet Griefing : This vulnerability is found in smart contract functions that

contain code to send Ethers to other smart contract accounts using the send function. The target smart contract can run a callback function when receiving Ethers and that function can contain improper code that makes the transaction run out of gas. The vulnerable function always runs out of gas and does not complete its work.

3. Integer Overflows : An example of this vulnerability is when a loop counter is uint8 and the terminating condition is more than 255. The code keeps looping infinite times since the value of uint8 is always between 0 to 255.

4. Possible Attacks and Incentives : These are possible attacks that need funds to exploit the vulnerabilities. These attacks include dumping Ether price, blackmailing of smart contract holders, or defaming competitors' smart contracts.

MadMax is a static program analysis technique that automatically detects gas-focused vulnerabilities. It uses a control-flow-analysis-based decompiler named Vandal to analyze the gas-focused vulnerabilities. The Vandal decompiler uses EVM bytecode as input and returns standard structured intermediate representation including a control-flow graph, three-address code for all operations, and recognized function boundaries. Then, Madmax analyzes the output from the Vandal decompiler using logic-based specifications. The analysis starts from the three-address code representation from the Vandal decompiler. Loops, induction variables, and data flow in smart contracts are first analyzed. Next, memory and dynamic data are analyzed to find smart contracts whose storage increases due to re-entry or nested arrays. Finally, the concept of gas-focused vulnerabilities is inferred. MadMax analyzes gas-focused vulnerabilities in 6.33 million smart contracts and found out that 4.1% of smart contracts have unbound iteration, 0.12% have Wallet Griefing, and 1.2% have integer overflow vulnerabilities.

3.2 Making Smart Contracts Smarter [2]

A smart contract is a program that is executed on a blockchain enforced by a consensus protocol. Smart contracts can implement a wide range of applications such as token transfer, financial derivatives, and savings. Since smart contracts typically store a large number of Ethers or tokens, many adversaries are attracted to these high value incentives and try to manipulate smart contract execution. Unfortunately, there is a lack of attention in the security of smart contracts, resulting in thousands of dollars' worth locked away in smart contracts that cannot be patched. All smart contracts must be checked for correctness before being deployed to the network. This related work identifies new security flaws in smart contracts which can be attacked by adversaries and well-known problems such as exceptions and logical flaws as follows:

1. Transaction-Ordering Dependence (TOD) : Since blockchain state is updated every time transactions are validated, the order of transactions is an important factor for the state update. For example, two users send transactions to withdraw all remaining funds in a smart contract at the same time. Only the transaction that is validated first is able to withdraw the fund, while the other transaction results in failure. Adversaries can use this flaw to attack target smart contracts by listening to all transactions calling the smart contract. If there are transactions worth attacking, adversaries send a transaction with the same input but set a higher *gas_price* so that the malicious transaction is validated before the normal transaction.

2. Timestamp Dependence : In some functions in smart contracts, block timestamp is used as an input to update their storage or used as a source for giving out rewards. Adversaries can participate in the network as miners and collect their malicious transactions in a block. Then, they publish the block when the block timestamp makes the malicious transactions benefit them.

3. Mishandled Exceptions : This flaw occurs when a smart contract calls a send

function which is used to send ether to another address without handling the result from sending. The target of the send may call some function and run out of gas while the main execution does not handle this failure. So, the execution continues even if failure occurs.

4. Re-entrancy Vulnerability : This is a well-known vulnerability that adversaries create a smart contract with a fallback function that calls the withdrawal function of the target smart contract. When the target smart contract sends Ethers to the malicious smart contract, a loop for calling the withdrawal function begins. As a result, the target smart contract sends most of its Ethers to the malicious smart contract.

This related work proposes Oyente, a tool with symbolic execution-based for analyzing the above flaws in smart contract source code. Oyente uses smart contract bytecode and global state to analyze the smart contract flaws. The result of the analysis are security problems of the smart contract control-flow graph. Oyente identifies at least one security problem from 8,833 out of 19,366 analyzed smart contracts. The flaws distribution is 5,411 (27.9%) smart contracts with Mishandled Exceptions, 3,056 (27.9%) smart contracts with TOD, 83 (0.94%) smart contracts with Timestamp Dependence, and 340 (3.85%) smart contracts with Re-entrancy Vulnerability.

3.3 SmartCheck: Static Analysis of Ethereum Smart Contracts [3]

SmartCheck focuses on security issues due to unfamiliar execution environment, new software stack, limitation on updating smart contract, anonymous attackers, rapid pace of development, and sub-optimal high-level language while assuming correctness of the Ethereum core infrastructure. Issues in smart contract source code are classified into four types.

1. Security issue : An issue that a malicious user can use for their own benefit. The examples of this are strict balance equality check, unchecked external call, re-entrancy, and the use of malicious third-party libraries.

2. Functional issue : This issue can cause a violation in the function called which can make smart contracts incompletely execute or cause the EVM to fail. Writing smart contracts without a withdrawal function and using improper type for variables can make functions incorrectly execute. Mathematical problem such as divide by zero, integer overflow, and integer underflow are issues that lead the EVM to fail.

3. Operational issue : This issue causes the EVM to have bad performance such as using bytes instead of byte array and creating loops with high gas-cost function inside.

4. Development issue : This issue occurs when smart contract source code is complex and difficult to comprehend. Examples are flexible compiler version, private modifier, using improper code style, and implicit function visibility.

SmartCheck is a static analysis tool implemented in Java which detects issues in smart contract source code written in Solidity. SmartCheck works by translating Solidity source code into an XML-based intermediate representation (IR) using ANTLR and a custom Solidity grammar. Then, vulnerability patterns are detected using XPath queries on the IR. 4,600 smart contracts was evaluated on SmartCheck. SmartCheck shows that 99.9% of smart contracts have at least one issue and 63.2% of smart contracts have critical issues. The most common issue detected by SmartCheck is implicit visibility level which is found 81,160 times.

The related work focuses on analyzing smart contract vulnerabilities and security issues. These related works require smart contract source code to perform the static analysis. Our work focuses on runtime errors from the EVM and analyzes them to see transaction behaviour without using smart contract's source code. In addition, our work helps users avoid sending transaction that are likely to fail.

Chapter IV

DESIGN

The main difference between our contribution and the related work is that our work focuses on runtime errors of failed transactions. There are 7 types of failed transaction that are thrown by the EVM as described in Section 2.5. We first identify runtime errors and estimate the monetary waste from failed transactions including transaction fees and block rewards. To reduce further failed transactions and monetary waste, we propose Evitar, a warning algorithm for reducing Ethereum smart contract runtime errors. Since Ethereum itself cannot prevent the occurrence of failed transactions, Evitar tries to prevent them by warning users not to send transactions that are likely to fail. An overview of how Evitar works, including the components of Evitar, its algorithm to reduce failed transactions, and how to use Evitar are described in this chapter.

4.1 Runtime error identification

In the Ethereum blockchain, we check a transaction's result using the status in the transaction receipt. However, the status only shows that a transaction is successful or not without showing how much progress was made during execution. When a transaction fails, a user cannot prevent further transactions on the same contract from failure since the runtime error of a failed transaction is unknown. In this section, we introduce the transaction trace that can be used to identify the runtime error of a transaction. A transaction trace is a record that stores all information of the transaction during its execution including the output state, amount of gas used and the runtime error. The transaction trace is not stored directly in the blockchain database but it can be obtained by requesting an Ethereum client such as Geth [37] and Parity [38] to re-execute the transaction again. To re-execute a transaction, the Ethereum client receives a transaction hash as input and queries global state before

transaction execution. Then, the Ethereum client re-executes the transaction and returns the transaction trace as a result. Since a transaction re-execution requires global state, only clients running as archive nodes are able to re-execute transactions. An archive node is a node that stores everything since the blockchain started. The data stored includes blocks, transactions, and all historical state. We compare transaction traces from Geth and Parity clients to determine the suitable client to use for identifying the runtime error of a transaction. Finally, we propose constructing trace trees, built from transaction traces obtained from the Parity client, for identifying the runtime error of a transaction.

Geth Trace and Parity Trace Comparison

Transaction traces from Geth and Parity clients have different formats due to their implementation. We compare the transaction trace format from both clients. The client with a more readable format is selected to use as the source for runtime error identification.

For the Geth client, the transaction trace is a list of EVM opcodes. The trace stores every instruction code executed for the transaction. Figure 4.1 is the trace structure from the Geth client. It shows instruction code information which contains the program counter (*pc*) to represent the order of the opcode, the opcode called (*op*), the gas cost to execute the opcode (*gasCost*), the remaining gas (*gas*), the current depth of the execution (*depth*), the error message (*error*), and the current state of stack, memory, and storage. Figure 4.2 shows a portion of the transaction trace, when the EVM executes the *DELEGATECALL* opcode, jumps to a lower depth, resets the *pc*, and continues the execution. The EVM jumps back to a higher depth when the *RETURN* opcode is executed. In addition, the EVM can inherit an error from a lower depth. The trace provided by the Geth client is difficult to comprehend since it is in the form of an instruction list, not easily human readable.

On the other hand, the Parity client provides more readable traces with many trace APIs such as *trace_call* for tracing a single call, *trace_callMany* for more

```
{
  "pc":      48,
  "op":      "DIV",
  "gasCost": 5,
  "gas":     64532,
  "depth":   1,
  "error":   null,
  "stack": [
    "00000000000000000000000000000000000000000000000000000000ffffffff",
    "0000000100000000000000000000000000000000000000000000000000000000",
    "2df07fbaabbe40e3244445af30759352e348ec8bebd4dd75467a9f29ec55d98d"
  ],
  "memory": [
    "0000000000000000000000000000000000000000000000000000000000000000",
    "0000000000000000000000000000000000000000000000000000000000000000",
    "0000000000000000000000000000000000000000000000000000000000000060"
  ],
  "storage": {
  }
}
```


Figure 4.1: Trace structure from tracing a transaction using the Geth client.

Step	PC	Operation	Gas	GasCost	Depth
[126]	392	DUP5	133867	3	1
[127]	393	ADD	133864	3	1
[128]	394	DUP6	133861	3	1
[129]	395	GAS	133858	2	1
[130]	396	DELEGATECALL	133856	131806	1
[131]	0	PUSH1	129206	3	2
[132]	2	PUSH1	129203	3	2
[133]	4	MSTORE	129200	12	2
[134]	5	PUSH1	129188	3	2

Figure 4.2: An example of Geth transaction trace.

than one trace in the same block, *trace_rawTransaction* for tracing a transaction by re-executing the transaction using a raw transaction as a new transaction and *trace_replayTransaction* for replaying a transaction again and return its trace as a result. In this thesis, *trace_replayTransaction* is used since it contains important in-

formation such as the error of the trace. Figure 4.3 shows the Parity trace structure from the *trace_replayTransaction* API. The trace returned from the Parity client consists of many important fields such as action that contains the transaction input provided by the user (*action*), transaction block identifiers (*blockHash* and *blockNumber*), execution result of a transaction trace (*result*), number of subtraces that a transaction trace called (*subtraces*), address of a transaction trace (*traceAddress*), transaction's hash (*transactionHash*), position of a transaction in a block (*transactionPosition*), and transaction type which can be "create" or "call" (*type*). The *action* field contains the call type such as CALL, CALLCODE and DELEGATECALL (*callType*), trace caller (*from*), amount of remaining gas (*gas*), calling input (*input*), target address for calling (*to*), and amount Ethers sent (*value*). The *result* field shows the amount of (gas used in this trace *(gasUsed)*, and the output return from this trace *(output)*.


```
{
    "action": {
        "callType": "call",
        "from": "0x676a07d53f837002b064272d6ab7da4af898e134",
        "gas": "0x25a7d",
        "input": "0x8f6ede1",
        "to": "0xb983e01458529665007ff7e0cddecdb74b967eb6",
        "value": "0x2b5e3af16b1880000"
        },
    "blockHash": "0x103f239550e61e17e9dc1eebb78381aaefc...",
    "blockNumber": 12825194,
    "result": {
        "gasUsed": "0x22367",
        "output": "0x000000000000000002a578919f7cdf12a7..."
    },
    "subtraces": 1,
    "traceAddress": [],
    "transactionHash": "0xbd36af0b348b26c94bd894b5725c0f...",
    "transactionPosition": 142,
    "type": "call"
}
```

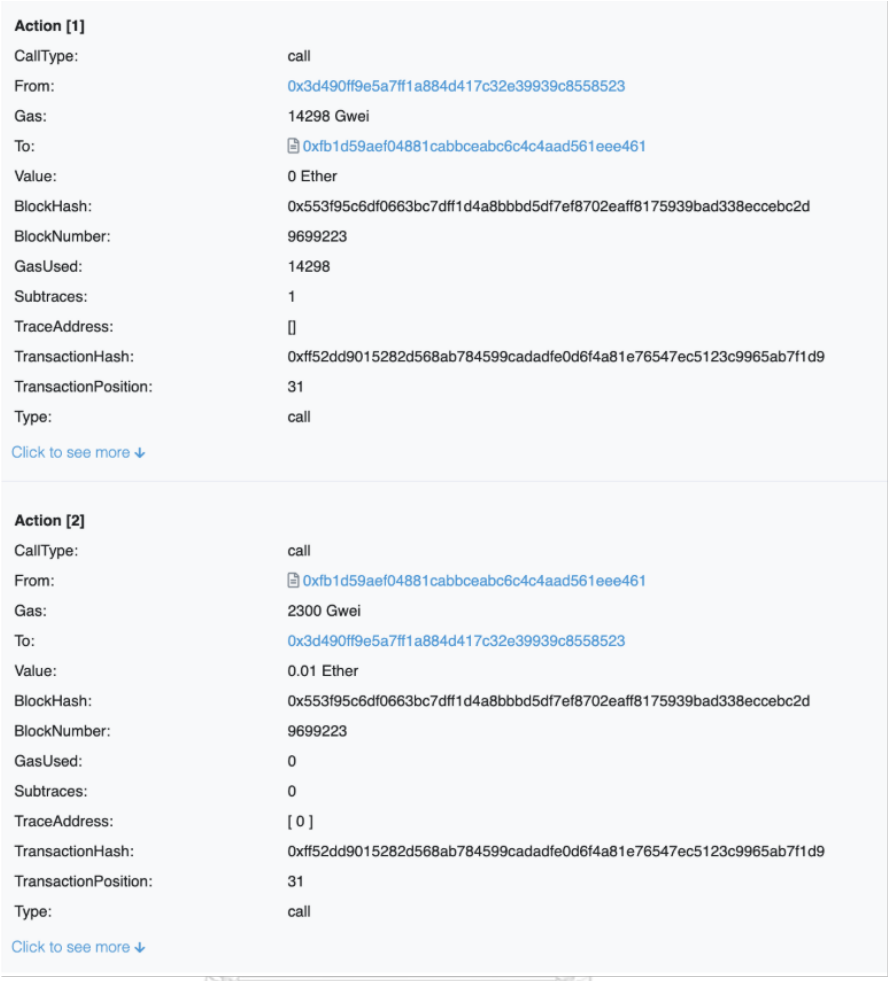

Figure 4.3: Transaction trace structure from Parity.

Parity Trace Extraction

We use the Parity client since it provides a more readable trace than the Geth client. Transaction traces from the Parity client are used to identify runtime errors of transactions. Figure 4.4 shows an example of a transaction trace extracted from the Parity client to create a trace tree. To enable the Parity client to run the *trace_replayTransaction* API, the Parity node is configured to run as a archive node using the following flags:

- *pruning* : this flag must be set to ”archive” to give the Parity nodea permission to maintain all the state in the state-trie. The state is necessary for tracing transactions because replaying transactions requires the knowledge of the state at the time the transaction was executed. Without the *pruning* flag set to archive, the Parity node stores only the state produced from the last few blocks and prunes out the old state.
- *fat-db* : this flag must be set to “on” so that the Parity node can store additional information such as the list of all accounts and their storage keys. Enabling the *fat-db* flag results in double the storage size on the node.
- *tracing* : this flag must be set to “on” to enable the Parity node get traces of each transaction from the EVM.

These three flags make Parity generates transaction traces. However, the drawback is that node storage grows extremely large compared to the default node. Figure 4.5 represents the growth of database for the default node and archive node. The archive node use around 14-16 times more storage than the default node. With these flags set, the Parity node is ready for tracing transactions in the blockchain. The next step for obtaining transaction traces is to extract necessary data from the Parity node. Since this thesis focuses on smart contract transactions and ignores Ether transfer transactions, all smart contracts in the blockchain are extracted first. Next, smart contract *called* transactions and their receipt are extracted by checking



Action [1]

CallType:	call
From:	0x3d490ff9e5a7ff1a884d417c32e39939c8558523
Gas:	14298 Gwei
To:	0xfb1d59aef04881cabbceabc6c4c4aad561eee461
Value:	0 Ether
BlockHash:	0x553f95c6df0663bc7dff1d4a8bbbd5df7ef8702eaff8175939bad338eccebc2d
BlockNumber:	9699223
GasUsed:	14298
Subtraces:	1
TraceAddress:	[]
TransactionHash:	0xff52dd9015282d568ab784599cadadfe0d6f4a81e76547ec5123c9965ab7f1d9
TransactionPosition:	31
Type:	call

Click to see more

Action [2]

CallType:	call
From:	0xfb1d59aef04881cabbceabc6c4c4aad561eee461
Gas:	2300 Gwei
To:	0x3d490ff9e5a7ff1a884d417c32e39939c8558523
Value:	0.01 Ether
BlockHash:	0x553f95c6df0663bc7dff1d4a8bbbd5df7ef8702eaff8175939bad338eccebc2d
BlockNumber:	9699223
GasUsed:	0
Subtraces:	0
TraceAddress:	[0]
TransactionHash:	0xff52dd9015282d568ab784599cadadfe0d6f4a81e76547ec5123c9965ab7f1d9
TransactionPosition:	31
Type:	call

Click to see more

Figure 4.4: An example of Parity transaction trace including its subtrace.

that *to_address* field is in the smart contract list or not. Then, transactions' hash are used as input for extracting transaction traces. Transaction traces are grouped by transaction hashes and are used to build a trace tree to identify the runtime error. Finally, the transaction status is checked with the runtime error to confirm whether or not the transaction failed. Some transactions contain subtraces that fail but do not result in the transaction itself failing.

Trace Tree

The trace tree is an n-ary tree that shows the transaction execution steps. A node in the trace tree represents the trace execution result and each trace can trigger other traces using *CALL*, *CALLCODE*, and *DELEGATECALL* opcodes. The traces

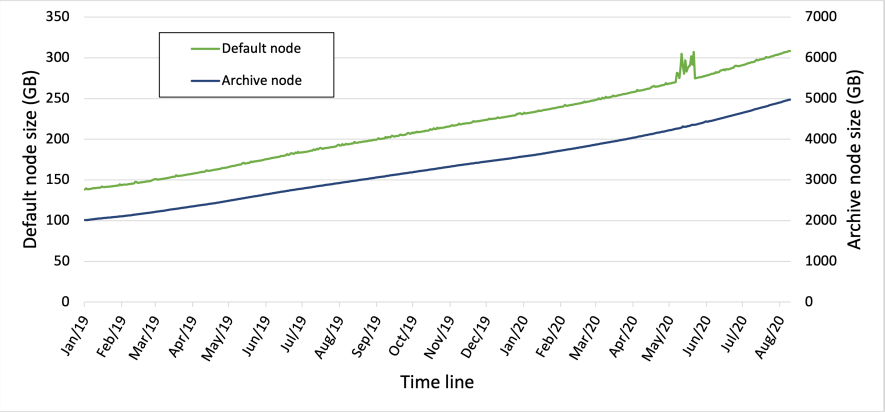
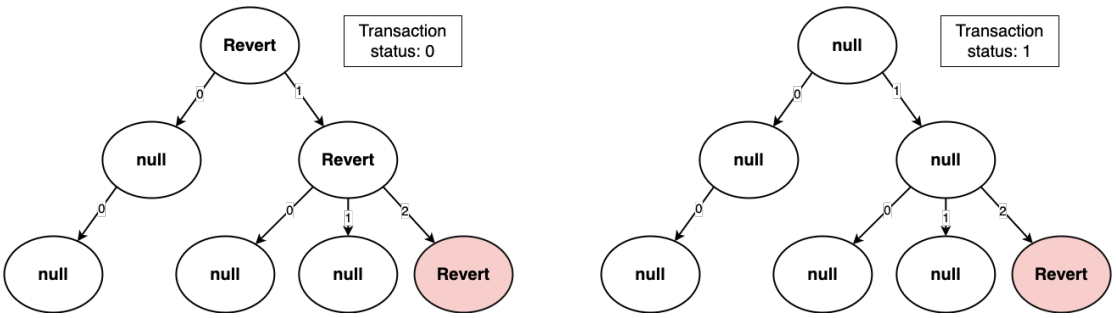


Figure 4.5: The growth of the database for Parity default node and Parity archive node.

called by their parent trace are displayed as childs node and are connected to their parent node ordered by the called time. There are several fields in the transaction trace required to build a trace tree such as *trace_address* and *error*. *Trace_address*, a string list, is used to locate the position of a node in the trace tree. Length of *trace_address* list refers to the depth of a trace tree while the value refers to the position of a node and its parent nodes. Error is the execution result of that trace. The trace tree is built by gathering all traces of a transaction. The starter node is a trace with no *trace_address* value (null value) and the value of a node is the runtime error from the trace (null value in *error* field means that trace is executed successfully). Then, other subtraces are connected to their parent. When all subtraces are connected together, the trace tree is formed and ready to identify the runtime error of the transaction. After the runtime error is identified, the *transaction_status* from transaction receipt is used to check if a transaction resulted in an error or not. As shown in Figure 4.6(a), the trace tree shows a Revert error in branch 1-2 and the transaction status is zero. So, their upward parents also resulted in failure and the transaction is marked as Revert. However, in Figure 4.6(b) the trace tree is the same as the previous example, but the transaction status is one. This means that the transaction is successful and the error at this child node does not result in the transaction failing.



(a) Trace tree with transaction status equal to 0.

(b) Trace tree with transaction status equal to 1.

Figure 4.6: Trace tree.

4.2 Monetary Waste from Failed Transactions

In this section, we estimate the monetary waste from failed transactions distributed by runtime errors. We will use the trace tree to identify the error distribution of the Ethereum network. The waste from failed transactions is calculated from wasted transaction fees and block rewards measured in both Ethers and dollars (using the daily exchange rate). The transaction fee is the amount of Ethers that is provided by users as a fee for miners who validate the transaction. Transaction fee waste is calculated by multiplying transaction gas used with the gas price set by the user at sending time. The block reward waste is calculated from number of the blocks needed to store these failed transactions multiplied by the amount of the block reward. In the Ethereum blockchain, the block reward is set to 5 Ethers per block. There have been two block reward reductions since the genesis block. The first reduction was at the Byzantium HF at block 4,370,000, when the block reward was decreased to 3 Ethers per block. The second reduction was at the Constantinople HF at block 7,280,000, when the block reward was decreased to 2 Ethers per block.

In this work, we measure waste from block 4,370,000, at the Byzantium HF when transaction status was first enabled, to block 10,600,000, a rounded-up block number of the experiment date (August 5, 2020). During this period, there are 425.2 million smart contract-related transactions with 18.9 million (4.45%) failed trans-

actions. We present the distribution of runtime errors for smart contract-related transactions in Figure 4.7 in log scale. The figure shows that the revert error is the most common runtime error compared to other runtime errors, followed by Out-of-Gas, Invalid-Opcode, Invalid-Jump, Stack-Underflow, Mutable-Call-In-Static-Context, and Stack-Overflow. The revert error is the most common because it is used to check if conditions meet the requirements of the smart contract function. Table 4.1 represents the transaction fee waste from failed transactions including gas used, and transaction fee in Ethers and dollars. The waste from the transaction fee is around 30,430 Ethers or $13 million if it is converted to dollars using the daily exchange rate.

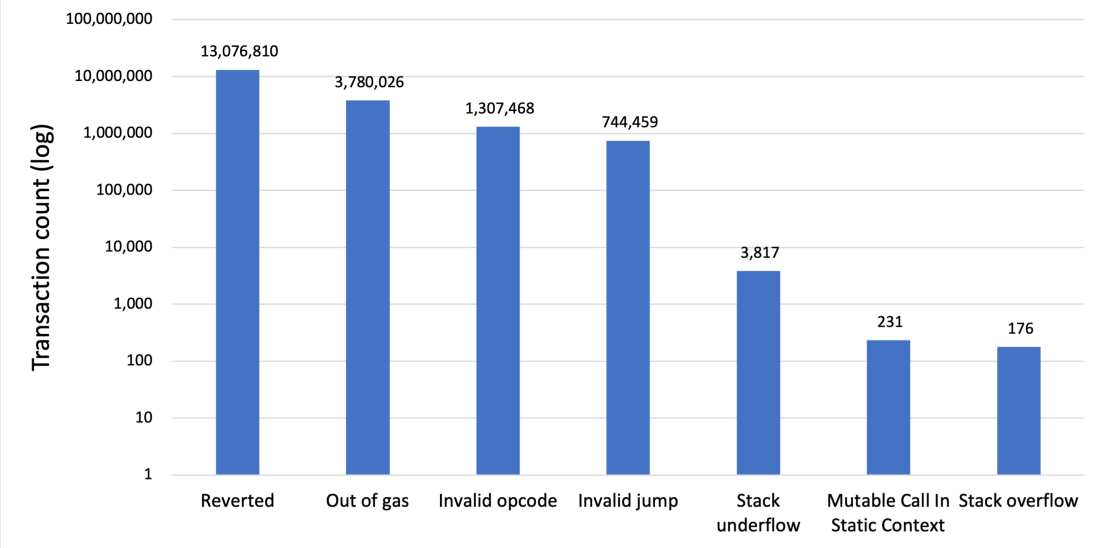


Figure 4.7: Distribution of runtime errors for smart contract transactions.

We estimate the block reward waste from the amount of gas used by failed transactions. These failed transactions need around 0.78 million blocks (387,575 Byzantium block and 396,510 Constantinople block) to store them. These blocks generate Ethers as block rewards equal to 1.96 million Ethers ($621.2 million). The total monetary waste from failed transactions is 2 million Ethers or $634.2 million.

Table 4.1: Transaction fee waste from failed smart contract transactions.

Runtime error	**Gas used (million)**	**Transaction fee (Ethers)**	**Transaction fee (Dollars)**
Reverted	575,593	12,399.39	4,237,397.34
Out of Gas	377,353	7,918.16	2,846,269.15
Invalid Opcode	265,429	5,410.07	2,496,913.07
Invalid Jump	179,917	4,625.18	3,428,937.86
Stack Underflow	3,671	68.82	43,748.41
Mutable Call in Static Context	488	6.19	1,141.38
Stack overflow	339	2.28	378.80
Total	1,402,791.39	30,430.08	13,054,786.01

These failed transactions cause users to pay a large amount of money for transaction fees without any benefit. In addition, these failed transactions are stored permanently in the blockchain and cannot be removed due to the immutable property of the blockchain. To prevent future monetary waste from failed transactions, transactions that are likely to result in failure must not be sent. Evitar helps users know that they are going to call a method with a high failure rate, so that they can avoid calling that method.

4.3 Evitar Overview

Evitar is an algorithm which dynamically analyzes mined transactions from the Ethereum blockchain and gives out a warning if the user requests for it. Figure 4.8 shows the overall workflow of Evitar. Evitar consists of three modules. First is the *TX-Processer* module which is a module for extracting data from the Ethereum blockchain, identifying runtime errors of transactions using the trace tree, and processing CSV files. Second is the *Warning algorithm* module, a module for analyzing mined transactions and marking a method status as a method with a high failure rate using two parameters: $windowsize(wnd)$ and $threshold(thresh)$. The Warning algorithm module detects methods with a high failure rate by analyzing, in real-time, a window of mined transactions of size wnd for each contract method called. After every wnd mined-but-not-yet-analyzed transactions have passed, the Warning algorithm module performs its analysis by analyzing the transaction error ratio observed at the end of the window. If the total error ratio is above the threshold $thresh$, the Warning algorithm module marks the method as a method with a high failure rate. Last is the *Server* module. This module stores the warning status from the *Warning algorithm* module and provides a RestAPI for users to check any method's status. In addition, Evitar suggests that users always send transactions with the maximum gas limit to avoid an Out of Gas error.

TX-Processer

Transactions, transaction receipts, and transaction traces of a method are used by Evitar to identify a warning status. This data is extracted from the Ethereum blockchain in this module. Then, we identify runtime errors of transactions using transaction receipts and transaction traces. Transactions and their runtime errors are the output of this module and are sent to the *Warning algorithm* module to check the warning status. There are two parts in this module: the Ethereum-ETL [39] and the Evitar Utils.

The Ethereum-ETL is a tool for extracting data from the Ethereum blockchain

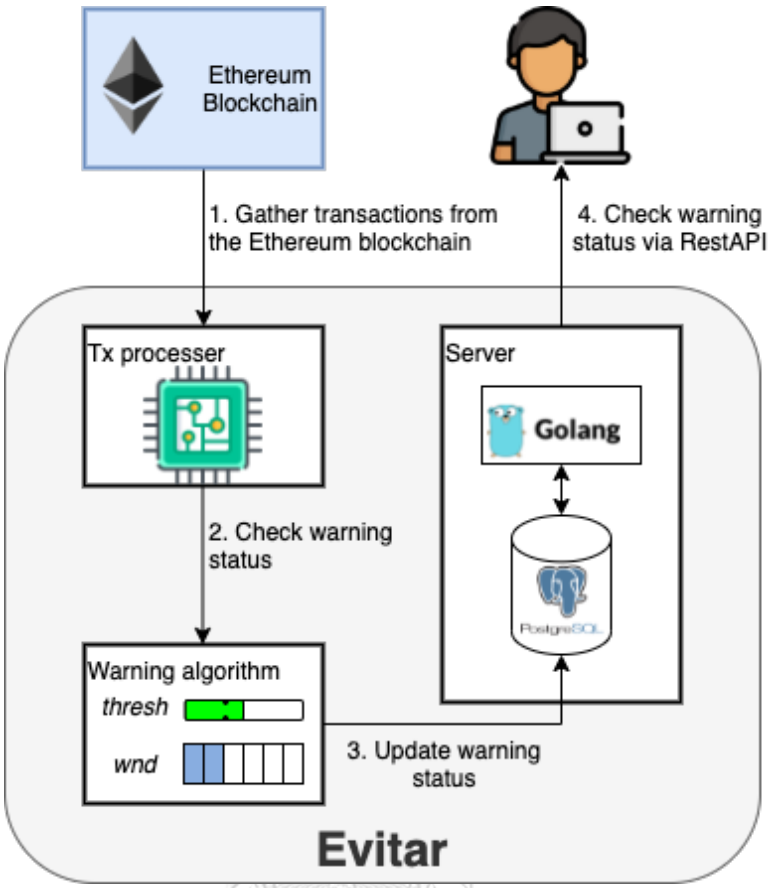


Figure 4.8: An overview of Evitar.

into a convenient format such as CSV files. This tool works by connecting to an Ethereum client to extract the blockchain data. The Ethereum-ETL can extract blocks, transactions, transaction receipts, transaction logs, transaction traces, as well as advanced data such as smart contracts, ERC20 [40, 41] and ERC721 [42, 43] tokens, and token transfers of which the schema are represented in Appendix B. The Ethereum network has two types of transactions: Ethers transfer transactions for sending Ethers between users and smart contract transactions for executing smart contracts. This module only focuses on smart contract transaction extraction especially smart contract *called* transactions which are used to execute smart contracts. A smart contract *created* transactions is sent only a single time per smart contract. For extracting transaction traces, the Ethereum client must run as an archive node. The Ethereum-ETL supports both Geth and Parity client and the exported schema are in the same format except transaction traces. Since we use transaction traces

from the Parity client, the Ethereum-ETL is used to extract data from the Parity client.

The Evitar Utils contains useful functions to process data for Evitar. There are three functions in the Evitar Utils as follows:

- *combine_csv* : a function for combining multiple csv files into a single file, mostly used when Ethereum-ETL is run in multiple batches and results in multiple files.
- *extract_csv_column* : a function for stripping out unnecessary columns such as *block_hash* and *block_timestamp*.
- *trace_runtime_error* : a function for identifying the runtime error of a transaction. This function receives transaction traces, generates a trace tree, and returns a runtime error as a result.

Warning Algorithm

The *Warning algorithm* module is a key module in Evitar since it analyzes the warning status to prevent future failed transactions. This module receives a newly mined transaction and its runtime error from the *Tx-Processer* module, then analyzes the transaction's method to see if it is a method with high failure rate or not by using two parameters: window size (wnd) and threshold ($thresh$).

- Window size (wnd) : a window of mined transactions of size wnd for each smart contract contract method called. After every wnd mined-but-not-yet-analyzed transactions have passed, we perform our analysis. We compute the transaction error ratio observed at the end of the window. Wnd is a positive integer. A high value of wnd makes the Warning algorithm wait too long before analyzing a method status. A low value makes the Warning algorithm analyzes a method status too frequently. For example, when the wnd is set to

1, a method is marked as a method with a high failure rate immediately if the first transaction failed.

- Threshold ($thresh$) : $thresh$ is the maximum error ratio acceptable for sending transactions to a method. If the error ratio for any type of error is above the threshold $thresh$, we mark that method as a method with a high failure rate and warn the user not to send the transaction. Otherwise, we suggest that users send the transaction using the maximum gas limit. $Thresh$ is a float value that can be set between 0 and 1. If $thresh$ is set with a high value, a low number of failed transactions can lead to the method being marked as having a high failure rate. However a low value of $thresh$ leads to the method not being marked as having a high failure rate even when there are a high number of failed transactions. A value of 1 marks a method as one with a high failure rate.

When this module receives a mined-but-not-yet-analyzed transaction from the *TX-Processer* module, it checks if the total mined-but-not-yet-analyzed transaction in the method reaches the wnd or not. If it does not reaches wnd, the method is ignored and waits for further transactions. Otherwise, this module compares the error ratio for any type of error with the $thresh$ value. If the error ratio is more than the $thresh$ value, the Warning algorithm marks that method as a method with a high failure rate. Then it updates the warning status of the method in the *Server* module.

Server

This module is an endpoint for users to check the warning status before calling that method. As shown in Figure 4.8, the *Server* module is implemented in the Go Programming Language (Golang) using PostgreSQL as a database. The Server's database contains three tables. The first table is the *evitar* table that stores Evitar configuration parameters with different $thresh$ and wnd values. Each Evitar configuration can result in different warning status for the same method. The second

table is the *method_status* table that stores transaction status such as the number of successful, reverted, and consumed-all-gas transactions, and the warning status of all methods in all smart contracts. This table also contains an *evitar* field which whether that the warning is the result from which Evitar configuration. The last table is the *method_runtime_error* that stores all runtime errors of all methods for all smart contracts. This table can be used to determine the runtime error distribution of any method. The *Server* module provides RestAPIs so that users can query a method's warning status and transaction behavior. Users can request for the available Evitar configurations parameters, all the methods in a smart contract, the warning status of a method in a smart contract, and the runtime error distribution of a method in a smart contract. The *Server*'s RestAPI specification is described in Appendix C.

Chapter V

EVALUATION

Evitar's performance is evaluated by extracting smart contract transactions from the Ethereum public network and replaying them in our Ethereum private network. We first evaluate the number of failed transaction reduced from warning against sending transactions. Then, the amount of saved gas consumption is estimated to measure Evitar's performance. Finally, we estimate the storage saved from sending transactions using *Evitar*.

5.1 Replay Transactions

To evaluate the performance of Evitar, we extract smart contract *called* transactions from the Ethereum public network. Next, we replay these transactions by sending them under Ethereum's default behavior and sending them using Evitar. Then, the number of failed transactions from both replays are compared to evaluate whether sending transactions with Evitar can reduce failed transactions. However, when replaying these transactions, we have to pretend to be the original transactions' senders. Senders' private keys are required since they are used to sign transactions before sending. However, private keys are kept secret by their owner. Because we do not have the access to the original sender's private key, we cannot replay transactions in the Ethereum public network.

Therefore, we replay transactions in an Ethereum private network instead. An Ethereum private network or private network is an isolated Ethereum network that runs the same protocol as the Ethereum public network but does not share the same database. In a private network, we change the original transactions' senders to our own accounts of which we have access to the accounts' private keys.

Setup Private network

Since a private network is isolated from the Ethereum public network, we can configure the settings to improve the replay performance. The Ethereum default configuration does not support replaying a lot of transactions in a short period. We modify the Ethereum client to have a constant block time and modify the genesis file to increase the block size. In addition, the genesis file is also modified to enable our private network to use the later hard fork features and initialize specific accounts with funds.

The Parity client is not integrated with a miner module, so we replay transactions using the Geth client that contains a miner module instead. The Parity client (archive node) is then synced with the Geth client (default node) after the replay is done to extract transaction data. We select the Geth client version 1.9.0-stable and the Parity client version 2.7.2-stable and modify the block difficulty calculation. The block difficulty depends on the time to generate a new block (block time). We set the block difficulty calculation to a constant value of 1,000,000 to fix the block time. As difficulty does not affect the logic of the smart contract execution, a static difficulty value is acceptable for reducing replay time.

We also need to modify the genesis file, a configuration file for generating the first block of the blockchain. There are three main purposes for modifying the genesis file. The first purpose is to increase the block gas limit which enables a block to store more transactions. Thus, the network can process a larger number of transactions per block during the replay than default. The second purpose is to supply initial funds to our accounts for replaying transactions. Because sending a transaction requires Ethers to pay the gas fee, initial funds allow us to skip the need to mine empty blocks to fund these accounts. The last purpose is to activate the later hard fork features such as transaction status in transaction receipt and revert error from the Byzantium HF at the first block in our private network instead of the much later real hard fork block. The fields modified in the genesis file are described as follows:

- *config* : This field contains the basic setting of the network. We set *chainId* to 1. We set *homesteadBlock*, *eip150Block*, *eip155Block*, *eip158Block*, *eip160Block*, and *byzantiumBlock* to 0. This configuration makes the Homestead HF, Byzantium HF and other EIP start at block 0. The last config, *daoForkSupport* is set to true to confirm that the node accepts the DAO HF.

- *alloc* : This field is used to allocate initial Ethers to specific accounts. There are two main purposes to fund accounts. The first is to initiate built-in accounts to replay smart contract transactions. The second purpose is to allocate Ethers to these accounts to use for creating and calling smart contracts. Each account is allocated 400 million Ethers to guarantee that there are sufficient Ethers for replaying transactions.

- *difficulty* : This field configures the difficulty of block 0. The difficulty value in the genesis file is important to an unmodified client since it is the starter difficulty of the network. A value too high can lead to long block time. This field is set to 0 to achieve the fastest block time.

- *gasLimit* : This field sets the maximum gas limit for blocks. GasLimit is the total maximum gas limit that all transactions inside a block are allowed to consume. A higher value of gasLimit allows a block to store more transactions. We set the value of this field to two billion to increase the block capacity and reduce the number of blocks generated during the replay. Increasing transactions per block can reduce the replay time, since in Evitar, a failed transaction that is sent with the maximum gas limit reduces the number of transactions per block to 1.

Starter Node Preparation

After the Ethereum client and the genesis file are modified, we can run a node to replay transactions. We deploy smart contracts to this node by using them as targets for smart contract *called* transactions. The node with smart contracts deployed is called the starter node.

To deploy smart contracts in the starter node, Tx-Processer is used to extract smart contract *created* transactions from the Ethereum public network. Then, only certain smart contract *created* transactions are replayed as follows. First, Tx-Processer extracts all smart contracts except for ERC20 and ERC721 smart contracts that are used for transferring digital tokens built on the Ethereum blockchain. The usage of token transfer is to send tokens between accounts by calling the smart contract and providing the receiver address in the transaction input. This means that the private key of the sender is required for signing and sending transactions, otherwise all token transfer transactions will logically result in failure. Obtaining those private keys for replaying transactions is impossible because they are privately stored by their owners. So we exclude these smart contracts from our evaluation. Next, we select smart contracts created by users and omit smart contracts created by other smart contracts because the original address of the smart contract is needed. Smart contracts created by users start with the "0x60" PUSH opcode, which is used for pushing variables into the stack for execution. Smart contracts that do not activate the Warning algorithm in Evitar are removed as they are called less than 100 times and have lower than 25% failure rate. After these steps, we obtain a list of smart contracts to replay and the corresponding created transactions. The number of contracts and transactions are shown in Table 5.1 After we filter smart contracts with these filter, we have 7,087 smart contracts to be deployed on the starter node.

Table 5.1: Number of smart contracts from block 1 to 10,600,000 selected for evaluation.

Smart contract Selection	**Number of smart contracts**
Non ERC20 and ERC721 smart contract	28,512,369
User created smart contract	2,816,204
Called > 100 times and failure rate ≥ 25% smart contract	7,087

After smart contract *created* transactions are extracted, they are deployed on the starter node using our account as the sender. We deploy smart contract *cre-*

ated transactions using Web3 python [44], a python library for interacting with the Ethereum client. The deployed smart contracts in our private network have different addresses compared to the original addresses since their addresses are randomly generated at the creation time. We keep a map of the new smart contract addresses and the original addresses and owners. This map is used as a target for smart contract *called* transactions to be sent. If these transactions are sent to their original addresses, the smart contract will not be executed since the addresses do not matched. When smart contracts are deployed and their addresses have been mapped, the starter node is ready to replay transactions. After smart contract *created* transactions are deployed, 5,953 out of 7,807 smart contracts are successfully deployed. The remaining fail due to specific conditions required in the source code such as the block number, address of creator, etc. A total of 72,056,308 transactions to these 5,953 smart contracts are extracted and are used as input for replaying transactions.

Transaction Input Modification

After smart contracts are deployed in the starter node and the transactions for replaying are extracted, we can replay transactions to evaluate Evitar's performance. To imitate the the Ethereum public network behavior, transactions are replayed in order using *block_number* and *transaction_index* of the original transactions. In addition, the original block gas limit of all blocks are also extracted from the Ethereum public network to use in our network to send transactions with the same maximum gas limit during replay. However, some transaction input are modified to make these transactions replayable in the private network as follows:

- *from_address* (modified) : The sender who signs and sends a transaction to the Ethereum network. This field is changed to our accounts since we do not have access to the original sender's private key. If the transaction sender is the smart contract owner, the *from_address* is changed to the account which deployed the smart contract. Otherwise, it is changed to one of our other

accounts.

- *to_address* (modified) : The smart contract address which is the target of the transaction. Since a smart contract address is a hash value that is newly generated when deployed, this field is changed to the contract's new address. We map the original smart contract address to the new smart contract address.

- *nonce* (modified) : This is a value that represents the number of transactions sent by the user. Changing this value affects the overall process. A transaction with a jump in *nonce* is a pending transaction that has to wait for a transaction with a lower *nonce* to be mined first. Transactions with a lower *nonce* cannot be sent to the network. So, we change this field to the nonce of our accounts.

- *gas* (modified) : The amount of gas provided to the EVM to execute the transaction. Since Evitar suggests users to send transactions with the maximum gas limit, this field is set to the maximum gas limit. The maximum gas limit for transactions can be found in the *gasLimit* field of the transaction's block.

- *gas_price* (modified) : The price that the user is willing to pay to the miner for each unit of gas. A higher value is a higher incentive for miners to select the transaction to mine. However, *gas_price* does not affect the replay process because transactions are replayed in their original order. So, it is set to a constant value.

- *input* (not modified) : The input of the transaction which is used by the EVM to execute a smart contract method. There are two types of inputs for smart contract-related transactions: input for creating and input for calling as mentioned in Section 2.4. The *input* of the transaction is not modified so we do not change the logic or the intention of the transaction during the replays.

- *value* (not modified) : The amount of Ethers that users provide in order to send to a smart contract or pay for something to reach a smart contract agreement. The *value* of transactions is not modified because of the same reason as *input*.

Evitar heuristics

Evitar proposes two heuristics to reduce the number of failed transactions in the Ethereum network. The first heuristics is termed *MaxGas*. *MaxGas* is based on simply sending all transactions using the maximum gas limit which enables the EVM able to continue its execution to reach a final outcome. This is a simple algorithm that can avoid most Out of Gas errors but has a drawback. The transaction may end up using more gas compared to sending transactions with its original gas value, if that transaction results in failure. The second heuristic is the Warning algorithm, an algorithm for analyzing the warning status of each smart contract method using *thresh* and *wnd*. We replay transactions with different heuristics to evaluate Evitar's performance as follows':

1. $Baseline$: This heuristic replays transactions without using the Warning algorithm and the maximum gas limit. The transactions sent by this heuristic use the original input, except for our emulated sender and receiver (smart contract) addresses.

2. $MaxGasOnly$: This heuristic replays transactions with $MaxGas$ but does not use the Warning algorithm. This heuristic is compared to the $Baseline$ heuristic to evaluate the performance of reducing Out of Gas errors.

3. $Evitar$: This heuristic uses $MaxGas$ along with the Warning algorithm to detect when to avoid sending transactions to methods using two parameters $thresh$ and wnd. If the Warning algorithm does not warn against sending transactions, they are sent with the maximum gas limit. To evaluate Evitar's performance, we run $Evitar$ with various parameters to identify the best parameter value.

   Replay transactions with each $Evitar$ configuration consumes a lot of computing resource and storage. So, we first run $Evitar$ in a smaller dataset to pre-evaluate Evitar's performance with different parameters configuration. Our small dataset consists of the top 500 called smart contracts that have more

than 25% failure rate from block 4,370,000 to block 8,650,000, and includes 2.6 million transactions that call them. To evaluate the parameter configuration, 3 settings with same $thresh$ value and 3 settings with same wnd value are used. In total, we run Evitar with 5 different parameter configurations as shown in Table 5.2 on our small dataset and the result is used to select Evitar's parameter configuration for running the full dataset. The results are shown in Table 5.2.

The result shows the number of successful and failed transactions of each $Evitar$ configuration. $Evitar_1$ ($thresh$=0.9), $Evitar_2$ ($thresh$=0.5), and, $Evitar_3$ ($thresh$=0.25) are used to compare the performance of $thresh$ when using a constant wnd value of 50. The result shows that a high $thresh$ value leads to a slight increase in successful transactions, but significant increase in failed transactions. A low $thresh$ value can reduce failed transactions, but trades off with significantly reduced successful transactions. So, $Evitar_2$ with 0.5 $thresh$ performs the best result among these 3 $Evitar$ settings. For the performance of wnd, we use $Evitar_2$ (wnd=50), $Evitar_4$ (wnd=25), and, $Evitar_5$ (wnd=10) for comparison and fix the $thresh$ value to 0.5. The result shows that lowering wnd can reduce failed transactions and retain more successful transactions. Our experiments show that $Evitar_5$ is the best parameter configuration for $Evitar$.

Table 5.2: Successful and failed transaction count (thousand) using Evitar with different $thresh$ and wnd values in our small dataset.

Evitar	wnd	thresh	Successful	Failed
$Evitar_1$	50	0.9	498.32	126.76
$Evitar_2$	50	0.5	486.99	88.46
$Evitar_3$	50	0.25	438.99	70.77
$Evitar_4$	25	0.5	486.92	68.94
$Evitar_5$	10	0.5	485.57	54.86

For the full dataset, we evaluate $Evitar$ with 3 different parameter configurations based on the results from these earlier experiments to evaluate the

performance of $thresh$ and wnd. The selected parameters for evaluation are shown as follows:

- $Evitar_2$: wnd = 50, $thresh$ = 0.5
- $Evitar_3$: wnd = 50, $thresh$ = 0.25
- $Evitar_5$: wnd = 10, $thresh$ = 0.5

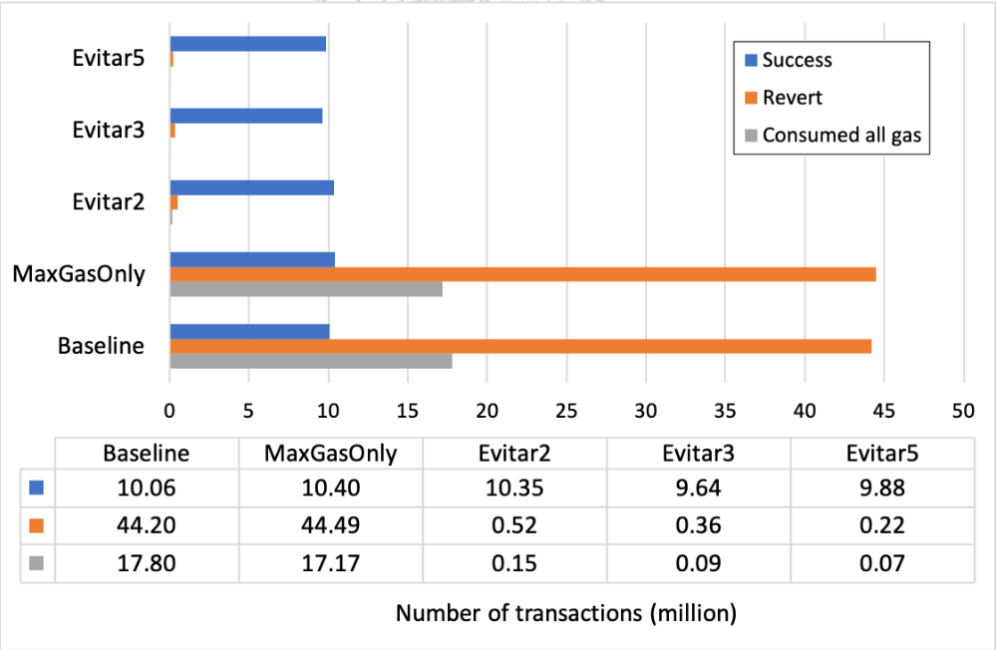
We omit $Evitar_1$ because $thresh$=0.9 resulted in 126.76 thousand failed transactions which is 38.3 thousand more failed transactions (43.30%) than $Evitar_2$. Despite that,the gain in the number of successful transactions is only from 486.99 to 498.32 (2.33%). We omit $Evitar_4$ because the result from $Evitar_5$ is more effective at reducing failed transactions. $Evitar_5$ has only 54.86 failed transactions which is 25.67% less than $Evitar_4$ while trading off with only 1.35 thousand (0.28%) fewer successful transactions.

5.2 Failed Transactions Reduction

We evaluate the reduction of failed transactions by comparing the number of transactions from each replay heuristic. We define three types of transaction status, success, revert, and consumed all gas. The first two transaction statuses directly refer to transactions that result in success and Revert error respectively. Consumed all gas status is a transaction that results in other runtime errors excluding the Revert error.

The results comparing the number of success, revert and consumed all gas from replaying transactions with all heuristics are shown in Figure 5.1. The number of successful transactions is roughly the same across all heuristics. The $MaxGasOnly$ heuristic sees a reduction in consumed all gas by 3.55% or around 0.63 million transactions compared to baseline, and 0.34 million transactions are converted into success while the rest resulted in revert. The result from $MaxGasOnly$ shows that sending transactions with the maximum gas limit can prevent gas exhaustion during smart contract execution. However, using $MaxGasOnly$ does not

always result in transaction success since other conditions may not be met during further execution. Meanwhile, using $Evitar_2$, the number of reverts is reduced from 44.20 million to 0.52 million (-98.82%) and consumed all gas is reduced from 17.80 million to 0.15 million (-99.14%). The number of successes increases from 10.06 million to 10.35 million (2.87%) compared to $Baseline$. This means that around 61.04 million transactions are prevented from being sent because they are detected as calls to methods with high failure rates. In addition, $Evitar_5$ can reduced more failed transactions compared to all other replay heuristics. It can reduce revert from 44.20 million to 0.22 million (-99.49%) and reduce consumed all gas from 17.80 million to 0.07 million (-99.60%) compared to $Baseline$. However, $Evitar_5$ also reduces the success rate from 10.06 million to 9.88 million (-1.78%). Evitar can prevent users from sending transactions to methods that are likely to fail.



	Baseline	MaxGasOnly	Evitar2	Evitar3	Evitar5
Success	10.06	10.40	10.35	9.64	9.88
Revert	44.20	44.49	0.52	0.36	0.22
Consumed all gas	17.80	17.17	0.15	0.09	0.07

Figure 5.1: Number of transactions from replaying transactions using Evitar different heuristics.

To be able to understand the impact of $thresh$ and wnd value better, we evaluate Evitar with three different parameter configurations. Table 5.3 shows the percentage of the amount of transaction changed in each transaction status of $Evitar_3$ and $Evitar_5$ compared to $Evitar_2$. In terms of $thresh$, the results from $Evitar_2$

($thresh$=0.5) and $Evitar_3$ ($thresh$=0.25) are used for comparison. Successful transactions are reduced by 6.77% when we lower the $thresh$ value from 0.5 to 0.25. Revert transactions and consume all gas transactions are reduced by 31.61% and 42.73% respectively. The result is satisfying since lowering $thresh$ value can reduce a large number of failed transactions, trading off with a small reduction in successful transactions. For the wnd value, the results from $Evitar_2$(wnd=50) and $Evitar_5$(wnd=10) are used for comparison. $Evitar_5$ reduces reverts by 57.08% and consume all gas transactions by 53.68% trading off with only a 4.52% reduction of successful transactions. Lowering both wnd and $thresh$ can reduce failed transactions, trading off with a reduction in successful transactions However, the result from lowering wnd shows better performance compared to lowering $thresh$, since it can reduce more failed transactions and trades off with less successful transactions. $Evitar_5$ performs the best performance compared to other configurations of $Evitar$.

Table 5.3: The percentage of the transactions changed for $Evitar_3$ and $Evitar_5$ compared to $Evitar_2$.

Evitar Compared to Evitar$_2$ (wnd=50, thresh=0.25)	**Success**	**Revert**	**Consumed all gas**
Evitar$_3$ (wnd=50, thresh=0.25)	-6.77%	-31.61%	-42.73%
Evitar$_5$ (wnd=10, thresh=0.5)	-4.52%	-57.08%	-53.68%

We have shown in that $Evitar$ has the potential to reduce failed transactions without hurting successful transactions. The number of failed transactions prevented are the result from using both the warning algorithm and the $MaxGas$ heuristic. The next section will discuss the performance of the $MaxGas$ heuristic.

5.3 MaxGas performance

Evitar suggests users send transactions with $MaxGas$, if a method called is not a method with a high failure rate. This is to prevent transactions from resulting in

an Out of Gas error. However, Section 5.2 shows that the number of consumed all gas transactions is reduced by only 3.55% in $MaxGasOnly$ compared to $Baseline$. On the other hand, $Evitar_5$ can reduce consumed all gas transactions up to 99.60% compared to $Baseline$. In this section, consumed all gas transactions are distributed by their runtime errors. Then, the amount of Out of Gas error is used to evaluate the Out of Gas error saved using $MaxGas$ heuristic.

Table 5.4: Transaction error distribution of $Baseline$, $MaxGasOnly$ and $Evitar_5$, including successful transactions.

Runtime Error	**Baseline**	**MaxGasOnly**	**Evitar$_5$**
Success	10,056,781	10,398,002	9,877,454
Revert	44,193,674	44,487,717	224,005
Out of Gas	634,982	4,740	216
Invalid Opcode	8,283,817	8,272,809	68,147
Invalid Opcode	8,283,817	8,272,809	68,147
Invalid Jump	8,887,054	8,893,040	2,739
Stack Underflow	0	0	0
Mutable Call in Static context	0	0	0
Stack Overflow	0	0	0

Consumed all gas error refers to many types of errors including Out of Gas, Invalid Opcode, Invalid Jump, Stack overflow, Stack underflow and Mutable Call in Static Context. So, the number of consumed all gas transactions saved cannot be directly referred as the number of Out of Gas error saved. To evaluate $MaxGas$ heuristic's performance, we identify the runtime errors of consumed all gas transactions using the Trace tree. Table 5.4 shows the runtime error distribution of $Baseline$, $MaxGasOnly$, and $Evitar_5$. $Evitar_5$ is chosen instead of the other $Evitar$ setting as it has the best performance. The table shows that $MaxGasOnly$ can reduce Out of Gas error from 634,982 to 4,740 (99.25%) compared to $Baseline$. However, the reduction for other consumed all gas errors are considered insignificant as it only reduces around 0.07-0.13%. On the other hand, $Evitar_5$ significantly reduces all

type of errors since it prevents sending transactions to methods that have a high failure rate.

The number of transactions prevented and saved is not the only factor we used to evaluate Evitar's performance. Gas consumption is also an important factor to measure. The next section will show the Evitar's performance in saving gas consumption.

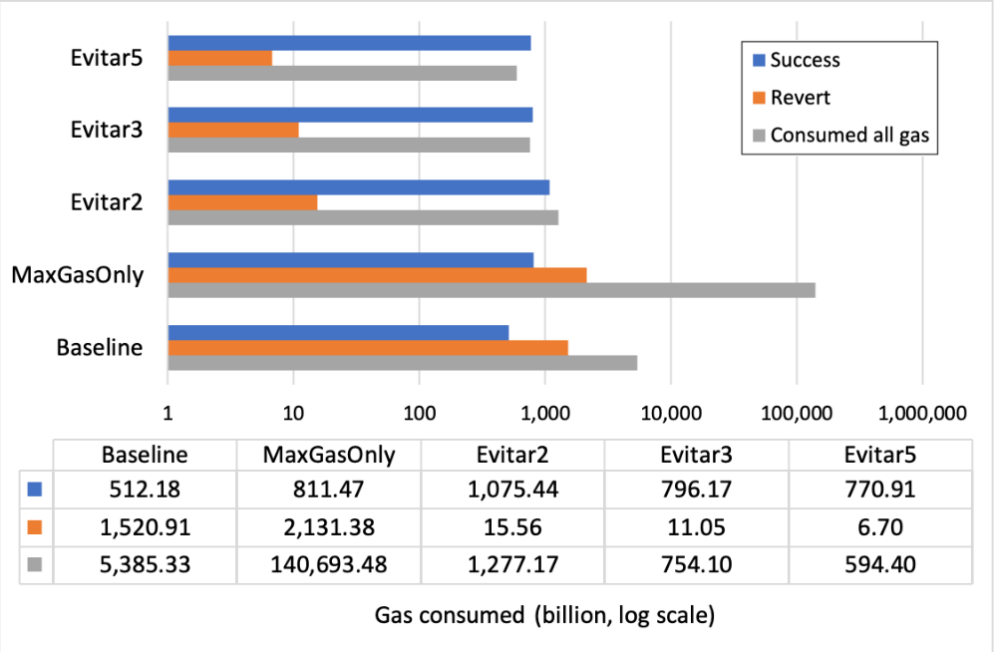
5.4 Gas Consumption Saved

Although Evitar can prevent failed transactions from being sent into the network, other transactions are still being sent with $MaxGas$. Without careful consideration, this may increase overall gas consumption. So, gas consumption is an important factor for evaluation. In this section, we evaluate the gas consumption of $Evitar$ in comparison to $Baseline$ and $MaxGasOnly$.

The total gas consumed by transactions for all transaction statuses across all heuristics are shown in Figure 5.2. Although the $MaxGasOnly$ heuristic can reduce the number of consumed all gas transactions compared to $Baseline$, but the amount of total gas used is significantly more than $Baseline$. Transactions with consumed all gas status in $Baseline$ consumes only 5,385.33 billion gas while $MaxGasOnly$ consumes 140,693.48 billion gas which is around 135,308.15 billion or 2,512.53% more than $Baseline$. For other transaction statuses, $MaxGasOnly$ gas consumption is also more than $Baseline$. There are 2131.38 billion reverts (58.43% increase), and 811 811.74 billion successes (40.14% increase) This result shows that $MaxGasOnly$ is the worst heuristic because it only reduces some consumed all gas errors in exchange for a large amount of gas used. This is because this type of error consumes all gas and we set the gas to maximum possible value based on the block gas limit resulting in more gas being consumed overall.

For $Evitar$, total gas consumption for revert and consumed all gas transaction

statuses are lower than $Baseline$. Compared to $Baseline$, the amount of gas consumed by $Evitar_2$ is 15.56 billion (-98.98%) reduction for revert and 1,277.17 billion (-76.28%) reduction for consumed all gas. For successful transactions, $Evitar$ has higher gas consumption at 1,075.44 (109.97% increase). The increase in gas consumption for successful transactions comes from some previously Out of Gas transactions that eventually changed into successful transactions. In addition, $Evitar_5$ consumed less gas than all other $Evitar$ settings. $Evitar_5$ consumes 6.70 billion gas in revert and consumes 594.40 billion gas in consumed all gas which are 99.69% and 88.96% less than $Baseline$, respectively. Evitar can prevent users from sending transactions that may result in failure and does not impact transaction processing performance like $MaxGasOnly$.



	Baseline	MaxGasOnly	Evitar2	Evitar3	Evitar5
■	512.18	811.47	1,075.44	796.17	770.91
■	1,520.91	2,131.38	15.56	11.05	6.70
■	5,385.33	140,693.48	1,277.17	754.10	594.40

Figure 5.2: The amount of gas consumed from replaying transactions using different heuristics.

In order to understand the impact of $thresh$ and wnd value on gas consumption Evitar with three different parameters is evaluated. Results from Table 5.5 show the percentage of gas consumption changed for $Evitar_3$ and $Evitar_5$ compared to $Evitar_2$. $Evitar_2$ ($thresh$=0.5) and $Evitar_3$ ($thresh$=0.25) are used to compare performance of $thresh$ value. Gas cost for successful transactions in $Evitar_3$ is 25.97%

lower than $Evitar_2$ due to the reduction in the amount of successful transactions. Gas consumption for revert and consumed all gas errors in $Evitar_3$ are reduced by 28.97% and 40.96% compared to $Evitar_2$. For the wnd value, we compare the results from $Evitar_2$(wnd=50) and $Evitar_5$(wnd=10). The gas consumption of success, revert and consumed all gas are reduced by 28.32%, 56.92% and 53.46% respectively when we lower the wnd value from 50 to 10. Lowering wnd can save more gas compared to lowering $thresh$ in all transaction statuses especially in revert transactions. This can concluded that $Evitar_5$ can save more gas compared to other configurations of $Evitar$.

Table 5.5: The percentage of gas consumption changed of $Evitar_3$ and $Evitar_5$ compared to $Evitar_2$.

Evitar Compared to Evitar$_2$ (wnd=50, thresh=0.25)	**Success**	**Revert**	**Consumed all gas**
Evitar$_3$ (wnd=50, thresh=0.25)	-25.97%	-28.97%	-40.96%
Evitar$_5$ (wnd=10, thresh=0.5)	-28.32%	-56.92%	-53.46%

$Evitar_5$ performs the best in term of failed transactions reduction and gas consumption saved compared to other configurations of $Evitar$. So, we can conclude that $Evitar_5$ with $thresh$=0.5 and wnd=10 is the best configuration.

5.5 Storage Saved

$Evitar$ can prevent users from sending transactions that are likely to fail. So, these transactions are not being stored in nodes resulting in less storage used. In this section, we calculate the storage saved in the Geth default node and the Parity archive node by comparing the node size of $Baseline$ and $Evitar_5$.

Figure 5.3 shows that node size of Geth default node and Parity archive node which are replayed with $Baseline$ and $Evitar_5$. For Geth default node, the node

size of $Baseline$ and $Evitar_5$ are 18.27 GB and 3.23 GB respectively which means that $Evitar_5$ uses 15.04 GB (82.32%) less storage compared to $Baseline$. For Parity archive node, node sizes of $Baseline$ and $Evitar_5$ are 34.02 GB and 16.98 GB respectively. So, $Evitar_5$ uses 17.04 GB (50.09%) less storage. We conclude that $Evitar_5$ can reduce the storage used in Ethereum nodes because $Evitar_5$ prevents users from sending transactions that are likely to fail. Because of that, Ethereum nodes store less failed transactions.

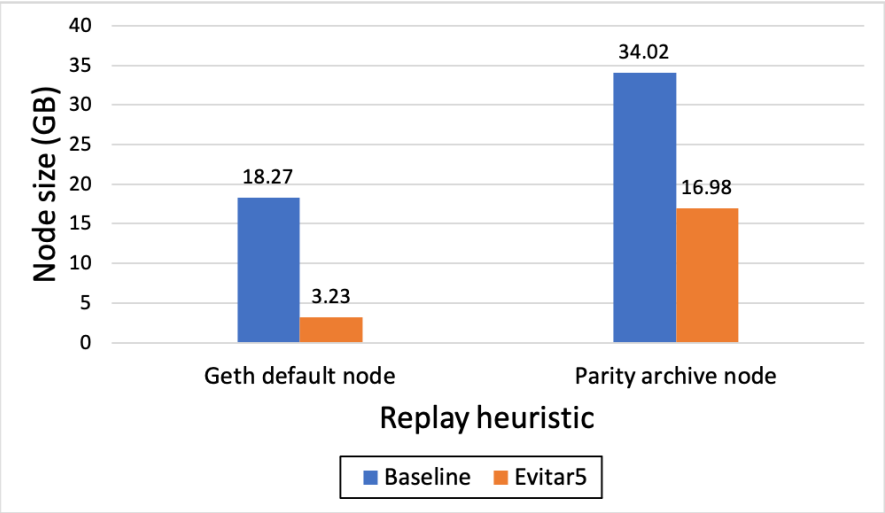


Figure 5.3: Size of Geth default node and Parity archive node replayed with $Baseline$ and $Evitar_5$ heuristics.

Storage saved estimation

To evaluate the amount of storage saved, we need to replay transactions and observe the actual node size which requires a significant amount of computation resources. Instead, we could also estimate the storage saved without replaying transactions by calculating the amount of transaction and gas consumption saved.

Estimated storage saved is the sum of the amount of blocks storage saved for blocks and transactions.

$$EstimatedStorageSaved = BlocksStorageSaved + TransactionsStorageSaved \quad (5.1)$$

The amount of storage saved for blocks is calculated by multiplying the num-

ber of blocks required to store all the transactions that were avoided (blocks saved) with the block size.

$$BlocksStorageSaved = BlocksSaved * BlockSize \tag{5.2}$$

Blocks saved can be computed as the ratio between the total gas saved and the block gas limit.

$$BlocksSaved = \frac{TotalGasSaved}{BlockGasLimit} \tag{5.3}$$

In general, transaction storage saved can be calculated by multiplying the number of transactions saved with the average transaction size (avgTxSize).

$$TransactionsStorageSaved = TransactionSaved * AvgTxSize \tag{5.4}$$

For example, we estimate the amount of storage saved using $Evitar_5$ compared to $Baseline$. First, we find the *BlocksSaved* and the *BlockSize* for calculating the *BlocksStorageSaved* shown in equation (5.2). The *BlocksSaved* is calculated from equation (5.3) using *TotalGasSaved* and *BlockGasLimit*. For the former, we use the total gas saved from Figure 5.2 which is 11.1 trillion gas saved. The latter is set to 12.5 million which is the block gas limit of block 10,600,000. So, the *BlocksSaved* is around 483.71 thousand blocks. Next, we find that the *BlockSize* is 538 bytes by mining an empty block in the private network. With the *BlocksSaved* and the *BlockSize*, the *BlocksStorageSaved* is 0.26 GB.

Second, the *TransactionsStorageSaved* is calculated from the *TransactionsSaved* and the *AvgTxSize* using equation (5.4). The former is the total transactions saved from Figure 5.1 which is around 61.9 million transactions saved. For the latter, we estimate the average transaction size using the block size and transactions per block from the Ethereum public network. As a result, the average transaction size is 180 bytes per transaction. This means that the *TransactionsStorageSaved* is around 11.14 GB.

Finally, we can calculate the *EstimatedStorageSaved* using *BlocksStorageSaved* and *TransactionsStorageSaved* as shown in equation (5.1). The *EstimatedStorageSaved* is 11.40 GB.

We compared the storage saved from the calculation with the actual storage saved of 15.04 GB as discussed in the previous section. The estimated storage saved is 3.64 GB less than the actual storage saved because the global state which is also stored in nodes is not calculated in our estimate. We can estimate the storage saved using the mentioned formula above along with the estimated number of transactions and total gas saved.

Chapter VI

CONCLUSION

6.1 Conclusion

Blockchain is an immutable distributed ledger technology that is disrupting computing in many sectors. However, its immutability presents new software engineering challenges in designing good code and executing successful transactions. Particularly, mined transactions that fail are processed no differently than successful transactions, this leads to a waste of computation, storage and transaction fees. So in this thesis, failed transactions that call smart contracts in Ethereum are dynamically analyzed to design a mechanism to avoid and reduce runtime errors, resulting in less overall system waste. This thesis proposes Evitar, an algorithm that dynamically analyzes transactions in the Ethereum network and gives out a warning status for smart contract methods that have a high failure rate. Evitar suggests users to avoid sending transactions to a method with a high failure rate. If a method has a high enough success rate, Evitar suggests sending transactions using the maximum gas limit. We evaluate Evitar's performance by replaying transactions in a private network with various heuristics. As a result, $Evitar_5$ can reduce revert and consumed all gas errors up to 99.49% and 99.60% respectively compared to $Baseline$, trading off with only a 1.78% reduction in successful transactions. For gas consumption, $Evitar_5$ consumes less gas than $Baseline$, saving 99.56% for revert and 88.96% for consumed all gas transactions. Next, sending transactions using $MaxGas$ can reduce Out of Gas errors by 99.25% compared to sending with normal gas. In addition, $Evitar_5$ can store less data compared to $Baseline$, up to 82.32% for the default node and 50.09% for the archive node.

6.2 Future work

In this thesis, we run Evitar with a selected set of $thresh$ and wnd values to evaluate Evitar's performance. However, other combinations of parameters may result in better performance than $Evitar_5$ ($thresh$=0.5 and wnd=10). Determining the combination of parameters that has the best performance further experiments with more combinations of $thresh$ and wnd value.

Next, Evitar suggests that users send transactions with the maximum gas allowed and avoid sending transactions to methods with high failure rates. Rather than using the maximum gas, one could try to send the optimal gas cost for each method. Sending transactions with the maximum gas limit can avoid insufficient gas problems during the EVM execution. However, this mechanism reduces the number of transactions processed per block to only one, if the transaction results in failure with any runtime error except the revert error. This is because the amount of gas consumed is equal to the block gas limit, so the block has no more space for other transactions. If the optimal gas cost is known, a transaction can be sent with lower gas compared to the maximum gas limit without impacting the transaction result. The optimal gas cost is the amount of gas that is exactly sufficient to make the EVM execute transactions successfully. However, it is difficult to estimate the optimal because methods in smart contracts work differently and the amount of gas consumed for each method is not equal. Each method must have its own optimal gas cost.

REFERENCES

[1] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, Y. Smaragdakis, MadMax: surviving out-of-gas conditions in Ethereum smart contracts, Proceedings of the ACM on Programming Languages 2 (OOPSLA) (2018) 1–27 (2018). doi:10.1145/3276486.

[2] L. Luu, D. H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, Proceedings of the ACM Conference on Computer and Communications Security 24-28-Octo (2016) 254–269 (2016). doi:10.1145/2976749.2978309.

[3] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, SmartCheck: Static analysis of ethereum smart contracts, Proceedings - International Conference on Software Engineering (October 2017) (2018) 9–16 (2018). doi:10.1145/3194113.3194115.

[4] F. Jake, Cryptocurrency Definition (2021).
URL https://www.investopedia.com/terms/c/cryptocurrency.asp

[5] V. Rao, Role of Cryptocurrency as means of exchange.
URL https://archive.telanganatoday.com/role-of-cryptocurrency-as-means-of-exchange

[6] N. Satoshi, Bitcoin: A Peer-to-Peer Electronic Cash System (2008).

[7] Cryptocurrency Prices, Charts And Market Capitalizations | CoinMarketCap.
URL https://coinmarketcap.com/

[8] VeChain Foundation, VeChain Whitepaper 2.0 (2019) 1–54 (2019).

[9] S. Tanwar, K. Parekh, R. Evans, Blockchain-based electronic healthcare record system for healthcare 4.0 applications, Journal of Information Security and Applications 50 (2020). doi:10.1016/j.jisa.2019.102407.

[10] G. O. Karame, E. Androulaki, 1984 - Stening, Everett - Response Styles (midpoint).pdf, Proceedings of the 2012 ACM conference on Computer and communications security (2012) 906–917 (2012).

[11] F. Jake, Double-Spending Definition (2020).
URL https://www.investopedia.com/terms/d/doublespending.asp

[12] G. Wood, Ethereum: a secure decentralised generalised transaction ledger, Ethereum project yellow paper 151 (2014) 1–32 (2014).

[13] V. Buterin, A next-generation smart contract and decentralized application platform, Etherum (January) (2014) 1–36 (2014).
URL http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf

[14] A. Savelyev, Contract law 2.0: 'Smart' contracts as the beginning of the end of classic contract law, Information and Communications Technology Law 26 (2) (2017) 116–134 (2017). doi:10.1080/13600834.2017.1301036.
URL https://doi.org/10.1080/13600834.2017.1301036

[15] M. P. Caro, M. S. Ali, M. Vecchio, R. Giaffreda, Blockchain-based traceability in Agri-Food supply chain management: A practical implementation, 2018 IoT Vertical and Topical Summit on Agriculture - Tuscany, IOT Tuscany 2018 (2018) 1–4 (2018). doi:10.1109/IOT-TUSCANY.2018.8373021.

[16] S. Wang, D. Li, Y. Zhang, J. Chen, Smart contract-based product traceability system in the supply chain scenario, IEEE Access 7 (2019) 115122–115133 (2019). doi:10.1109/ACCESS.2019.2935873.

[17] Q. Lin, H. Wang, X. Pei, J. Wang, Food Safety Traceability System Based on Blockchain and EPCIS, IEEE Access 7 (2019) 20698–20707 (2019). doi:10.1109/ACCESS.2019.2897792.

[18] P. McCorry, S. F. Shahandashti, F. Hao, A smart contract for boardroom voting with maximum voter privacy, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes

in Bioinformatics) 10322 LNCS (2017) 357–375 (2017). doi:10.1007/978-3-319-70972-7_20.

[19] F. P. Hjalmarsson, G. K. Hreioarsson, M. Hamdaqa, G. Hjalmtysson, Blockchain-Based E-Voting System, IEEE International Conference on Cloud Computing, CLOUD 2018-July (2018) 983–986 (2018). doi:10.1109/CLOUD.2018.00151.

[20] CryptoKitties | Collect and breed digital cats!
URL https://www.cryptokitties.co/

[21] A. Serada, T. Sihvonen, J. T. Harviainen, CryptoKitties and the New Ludic Economy: How Blockchain Introduces Value, Ownership, and Scarcity in Digital Gaming, Games and Culture 16 (4) (2020) 457–480 (2020). doi:10.1177/1555412019898305.

[22] E. Ordano, A. Meilich, Y. Jardi, M. Araoz, Decentraland white paper (2017) 15 (2017).

[23] Swipe Wallet, Venus - The Money Market & Synthetic Stablecoin Protocol (2020).

[24] H. Adams, N. Zinsmeister, M. Salem moody, uniswaporg River Keefer, D. Robinson, Uniswap v3 Core (March) (2021).

[25] S. Jumnongsaksub, K. Sripanidkulchai, Reducing Smart Contract Runtime Errors on Ethereum, IEEE Software 37 (5) (2020) 55–59 (2020). doi:10.1109/MS.2020.2993882.

[26] F. Jake, Distributed Ledger Technology (DLT) Definition (2021).
URL https://www.investopedia.com/terms/d/distributed-ledger-technology-dlt.asp

[27] K. Will, Merkle Root (Cryptocurrency) Definition (2020).
URL https://www.investopedia.com/terms/m/merkle-root-cryptocurrency.asp

[28] H. Luit, History of Ethereum Hard Forks.
URL https://medium.com/mycrypto/the-history-of-ethereum-hard-forks-6a6

[29] Ethereum Community, Ethereum Homestead Documentation, Github (2017) https://www.ethereum.org/ (2017).
URL http://www.ethdocs.org/en/latest/

[30] Ethereum Foundation, Byzantium HF Announcement | Ethereum Foundation Blog.
URL https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/

[31] Ethereum Foundation, Ethereum Constantinople/St. Petersburg Upgrade Announcement | Ethereum Foundation Blog.
URL https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/

[32] Ethereum Foundation, Ethereum Constantinople Upgrade Announcement | Ethereum Foundation Blog.
URL https://blog.ethereum.org/2019/01/11/ethereum-constantinople-upgrade-announcement/

[33] Ethereum Foundation, Ethereum Istanbul Upgrade Announcement | Ethereum Foundation Blog.
URL https://blog.ethereum.org/2019/11/20/ethereum-istanbul-upgrade-announcement/

[34] Little Jay, Guido Dan, Boukli-Hacene Omar, Nolan Dan, Pldespaigne, Ia, Schoe Afr, GitHub - crytic/evm-opcodes: Ethereum opcodes and instruction reference.
URL https://github.com/crytic/evm-opcodes

[35] Solidity — Solidity 0.8.4 documentation.
URL https://docs.soliditylang.org/en/v0.8.4/

[36] Vyper — Vyper documentation.
URL https://vyper.readthedocs.io/en/stable/

[37] Go Ethereum.
URL https://geth.ethereum.org/

[38] Parity Ethereum Client - OpenEthereum | Parity Technologies.
URL https://www.parity.io/ethereum/

[39] M. Evgeny, GitHub - blockchain-etl/ethereum-etl: Python scripts for ETL.
URL https://github.com/blockchain-etl/ethereum-etl

[40] EIP-20: ERC-20 Token Standard.
URL https://eips.ethereum.org/EIPS/eip-20

[41] ERC-20 Token Standard | ethereum.org.
URL https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

[42] EIP-721: ERC-721 Non-Fungible Token Standard.
URL https://eips.ethereum.org/EIPS/eip-721

[43] ERC-721 Non-Fungible Token Standard | ethereum.org.
URL https://ethereum.org/en/developers/docs/standards/tokens/erc-721/

[44] Introduction — Web3.py 5.19.0 documentation.
URL https://web3py.readthedocs.io/en/stable/

Appendix I

EVM GAS TABLE

Table A.1: EVM Opcode gas cost.

Opcode	Name	Description	Gas
0x00	STOP	Halts execution	0
0x01	ADD	Addition operation	3
0x02	MUL	Multiplication operation	5
0x03	SUB	Subtraction operation	3
0x04	DIV	Integer division operation	5
0x05	SDIV	Signed integer division operation (truncated)	5
0x06	MOD	Modulo remainder operation	5
0x07	SMOD	Signed modulo remainder operation	5
0x08	ADDMOD	Modulo addition operation	8
0x09	MULMOD	Modulo multiplication operation	8
0x0a	EXP	Exponential operation	(exp == 0) ? 10 : (10 + 10 * (1 + log256(exp)))
0x0b	SIGNEX-TEND	Extend length of two's complement signed integer	5
0x0c - 0x0f	Unused	Unused	
0x10	LT	Less-than comparison	3
0x11	GT	Greater-than comparison	3
0x12	SLT	Signed less-than comparison	3
0x13	SGT	Signed greater-than comparison	3
0x14	EQ	Equality comparison	3
0x15	ISZERO	Simple not operator	3
0x16	AND	Bitwise AND operation	3
0x17	OR	Bitwise OR operation	3
0x18	XOR	Bitwise XOR operation	3
0x19	NOT	Bitwise NOT operation	3
0x1a	BYTE	Retrieve single byte from word	3
0x1b	SHL	Shift Left	3
0x1c	SHR	Logical Shift Right	3
0x1d	SAR	Arithmetic Shift Right	3

Table A.2: EVM Opcode gas cost(cont).

Opcode	Name	Description	Gas
0x20	SHA3	Compute Keccak-256 hash	30 + 6 * (size of input in words)
0x21 - 0x2f	Unused	Unused	
0x30	ADDRESS	Get address of currently executing account	2
0x31	BALANCE	Get balance of the given account	700
0x32	ORIGIN	Get execution origination address	2
0x33	CALLER	Get caller address	2
0x34	CALLVALUE	Get deposited value by the instruction/ transaction responsible for this execution	2
0x35	CALLDAT-ALOAD	Get input data of current environment	3
0x36	CALLDATA-SIZE	Get size of input data in current environment	2 + 3 * (number of words copied, rounded up)
0x37	CALLDATA-COPY	Copy input data in current environment to memory	3
0x38	CODESIZE	Get size of code running in current environment	2
0x39	CODECOPY	Copy code running in current environment to memory	2 + 3 * (number of words copied, rounded up)
0x3a	GASPRICE	Get price of gas in current environment	2
0x3b	EXTCODE-SIZE	Get size of an account's code	700
0x3c	EXTCODE-COPY	Copy an account's code to memory	700 + 3 * (number of words copied, rounded up)
0x3d	RETURN-DATASIZE	Pushes the size of the return data buffer onto the stack	2
0x3e	RETURN-DATACOPY	Copies data from the return data buffer to memory	3
0x3f	EXTCODE-HASH	Returns the keccak256 hash of a contract's code	700
0x40	BLOCKHASH	Get the hash of one of the 256 most recent complete blocks	20
0x41	COINBASE	Get the block's beneficiary address	2
0x42	TIMESTAMP	Get the block's timestamp	2
0x43	NUMBER	Get the block's number	2
0x44	DIFFICULTY	Get the block's difficulty	2
0x45	GASLIMIT	Get the block's gas limit	2
0x46	CHAINID	Returns the current chain's EIP-155 unique identifier	2
0x47 - 0x4f	Unused	-	

Table A.3: EVM Opcode gas cost(cont).

Opcode	Name	Description	Gas
0x50	POP	Remove word from stack	2
0x51	MLOAD	Load word from memory	3*
0x52	MSTORE	Save word to memory	3*
0x53	MSTORE8	Save byte to memory	3
0x54	SLOAD	Load word from storage	800
0x55	SSTORE	Save word to storage	((value != 0) && (storage_location == 0)) ? 20000 : 5000
0x56	JUMP	Alter the program counter	8
0x57	JUMPI	Conditionally alter the program counter	10
0x58	GETPC	Get the value of the program counter prior to the increment	2
0x59	MSIZE	Get the size of active memory in bytes	2
0x5a	GAS	Get the amount of available gas, including the corresponding reduction the amount of available gas	2
0x5b	JUMPDEST	Mark a valid destination for jumps	1
0x5c - 0x5f	Unused	-	
0x60	PUSH1	Place 1 byte item on stack	3
0x61	PUSH2	Place 2-byte item on stack	3
0x62	PUSH3	Place 3-byte item on stack	3
0x63	PUSH4	Place 4-byte item on stack	3
0x64	PUSH5	Place 5-byte item on stack	3
0x65	PUSH6	Place 6-byte item on stack	3
0x66	PUSH7	Place 7-byte item on stack	3
0x67	PUSH8	Place 8-byte item on stack	3
0x68	PUSH9	Place 9-byte item on stack	3
0x69	PUSH10	Place 10-byte item on stack	3
0x6a	PUSH11	Place 11-byte item on stack	3
0x6b	PUSH12	Place 12-byte item on stack	3
0x6c	PUSH13	Place 13-byte item on stack	3
0x6d	PUSH14	Place 14-byte item on stack	3
0x6e	PUSH15	Place 15-byte item on stack	3
0x6f	PUSH16	Place 16-byte item on stack	3
0x70	PUSH17	Place 17-byte item on stack	3
0x71	PUSH18	Place 18-byte item on stack	3
0x72	PUSH19	Place 19-byte item on stack	3
0x73	PUSH20	Place 20-byte item on stack	3
0x74	PUSH21	Place 21-byte item on stack	3

Table A.4: EVM Opcode gas cost(cont).

Opcode	Name	Description	Gas
0x75	PUSH22	Place 22-byte item on stack	3
0x76	PUSH23	Place 23-byte item on stack	3
0x77	PUSH24	Place 24-byte item on stack	3
0x78	PUSH25	Place 25-byte item on stack	3
0x79	PUSH26	Place 26-byte item on stack	3
0x7a	PUSH27	Place 27-byte item on stack	3
0x7b	PUSH28	Place 28-byte item on stack	3
0x7c	PUSH29	Place 29-byte item on stack	3
0x7d	PUSH30	Place 30-byte item on stack	3
0x7e	PUSH31	Place 31-byte item on stack	3
0x7f	PUSH32	Place 32-byte (full word) item on stack	3
0x80	DUP1	Duplicate 1st stack item	3
0x81	DUP2	Duplicate 2nd stack item	3
0x82	DUP3	Duplicate 3rd stack item	3
0x83	DUP4	Duplicate 4th stack item	3
0x84	DUP5	Duplicate 5th stack item	3
0x85	DUP6	Duplicate 6th stack item	3
0x86	DUP7	Duplicate 7th stack item	3
0x87	DUP8	Duplicate 8th stack item	3
0x88	DUP9	Duplicate 9th stack item	3
0x89	DUP10	Duplicate 10th stack item	3
0x8a	DUP11	Duplicate 11th stack item	3
0x8b	DUP12	Duplicate 12th stack item	3
0x8c	DUP13	Duplicate 13th stack item	3
0x8d	DUP14	Duplicate 14th stack item	3
0x8e	DUP15	Duplicate 15th stack item	3
0x8f	DUP16	Duplicate 16th stack item	3
0x90	SWAP1	Exchange 1st and 2nd stack items	3
0x91	SWAP2	Exchange 1st and 3rd stack items	3
0x92	SWAP3	Exchange 1st and 4th stack items	3
0x93	SWAP4	Exchange 1st and 5th stack items	3
0x94	SWAP5	Exchange 1st and 6th stack items	3
0x95	SWAP6	Exchange 1st and 7th stack items	3
0x96	SWAP7	Exchange 1st and 8th stack items	3
0x97	SWAP8	Exchange 1st and 9th stack items	3
0x98	SWAP9	Exchange 1st and 10th stack items	3
0x99	SWAP10	Exchange 1st and 11th stack items	3
0x9a	SWAP11	Exchange 1st and 12th stack items	3

Table A.5: EVM Opcode gas cost(cont).

Opcode	Name	Description	Gas
0x9b	SWAP12	Exchange 1st and 13th stack items	3
0x9c	SWAP13	Exchange 1st and 14th stack items	3
0x9d	SWAP14	Exchange 1st and 15th stack items	3
0x9e	SWAP15	Exchange 1st and 16th stack items	3
0x9f	SWAP16	Exchange 1st and 17th stack items	3
0xa0	LOG0	Append log record with no topics	375
0xa1	LOG1	Append log record with one topic	750
0xa2	LOG2	Append log record with two topics	1125
0xa3	LOG3	Append log record with three topics	1500
0xa4	LOG4	Append log record with four topics	1875
0xa5 - 0xaf	Unused	-	
0xb0	JUMPTO	Tentative libevmasm has different numbers	
0xb1	JUMPIF	Tentative	
0xb2	JUMPSUB	Tentative	
0xb4	JUMPSUBV	Tentative	
0xb5	BEGINSUB	Tentative	
0xb6	BEGINDATA	Tentative	
0xb8	RETURNSUB	Tentative	
0xb9	PUTLOCAL	Tentative	
0xba	GETLOCAL	Tentative	
0xbb - 0xe0	Unused	-	
0xe1	SLOAD-BYTES	Only referenced in pyethereum	-
0xe2	SSTORE-BYTES	Only referenced in pyethereum	-
0xe3	SSIZE	Only referenced in pyethereum	-
0xe4 - 0xef	Unused	-	
0xf0	CREATE	Create a new account with associated code	32000
0xf1	CALL	Message-call into an account	Complicated
0xf2	CALLCODE	Message-call into this account with alternative account's code	Complicated
0xf3	RETURN	Halt execution returning output data	0
0xf4	DELEGATE-CALL	Message-call into this account with an alternative account's code, but persisting into this account with an alternative account's code	Complicated
0xf5	CREATE2	Create a new account and set creation address to sha3(sender + sha3(init code)) % 2**160	

Table A.6: EVM Opcode gas cost(cont).

Opcode	Name	Description	Gas
0xf6 - 0xf9	Unused	-	
0xfa	STATICCALL	Similar to CALL, but does not modify state	40
0xfb	Unused	-	
0xfc	TXEXECGAS	Not in yellow paper FIXME	-
0xfd	REVERT	Stop execution and revert state changes, without consuming all provided gas and providing a reason	0
0xfe	INVALID	Designated invalid instruction	0
0xff	SELFDE-STRUCT	Halt execution and register account for later deletion	5000 + ((create_new_account) ? 25000 : 0)

Appendix II

ETHEREUM-ETL SCHEMA

Table B.1: Block schema.

Column	Type
number	bigint
hash	hex_string
parent_hash	hex_string
nonce	hex_string
sha3_uncles	hex_string
logs_bloom	hex_string
transactions_root	hex_string
state_root	hex_string
receipts_root	hex_string
miner	address
difficulty	numeric
total_difficulty	numeric
size	bigint
extra_data	hex_string
gas_limit	bigint
gas_used	bigint
timestamp	bigint
transaction_count	bigint

Table B.2: Transaction schema.

Column	Type
hash	hex_string
nonce	bigint
block_hash	hex_string
block_number	bigint
transaction_index	bigint
from_address	address
to_address	address
value	numeric
gas	bigint
gas_price	bigint
input	hex_string
block_timestamp	bigint

Table B.3: Receipt schema.

Column	Type
transaction_hash	hex_string
transaction_index	bigint
block_hash	hex_string
block_number	bigint
cumula- tive_gas_used	bigint
gas_used	bigint
contract_address	address
root	hex_string
status	bigint

Table B.4: Contract schema.

Column	Type
address	address
bytecode	hex_string
function_sighashes	string
is_erc20	boolean
is_erc721	boolean
block_number	bigint

Table B.5: Trace schema.

Column	Type
block_number	bigint
transaction_hash	hex_string
transaction_index	bigint
from_address	address
to_address	address
value	numeric
input	hex_string
output	hex_string
trace_type	string
call_type	string
reward_type	string
gas	bigint
gas_used	bigint
subtraces	bigint
trace_address	string
error	string
status	bigint
trace_id	string

Appendix III

EVITAR API SPECIFICATION

Table C.1: ListEvitar API.

API Name	ListEvitar
Method	GET
URI	/list-evitar
Description	API for query list of all available Evitar with different thresh and wnd
Request params	-

Response body	{
	“evitar_list”: [
	{
	“evitar”: “evitar_1”,
	“thresh”: 0.5,
	“wnd”: 50
	},
	{
	“evitar”: “evitar_2”,
	“thresh”: 0.25,
	“wnd”: 50
	},
	]
	}

Table C.2: GetMethodsInSmartContract API.

API Name	GetMethodsInSmartContract
Method	GET
URI	/get-method-in-smart-contract
Description	API for query all available methods in the smart contract
Request params	1. smart_contract_address (string)

Response body	{
	“contract_address”: ”0x3D490ff9e5A7FF...”,
	“methods”: [
	“0xc12fba87”,
	“0x1b7faac3”
	],
	}

Table C.3: IsMethodWithHighFailureRate API.

API Name	IsMethodWithHighFailureRate
Method	GET
URI	/is-method-with-high-failure-rate
Description	API for checking the status of method in smart contract
Request params	1. address (string) 2. method (string) 3. evitar (string)

Response body	{
	“evitar”: “evitar_1”,
	“contract_address” : ”0x3D490ff9e5A7FF...”,
	“methods”: “0xc12fba87”,
	“method_with_high_failure_rated”: true,
	“successful”: 47,
	“revert”: 30,
	“consumed_all_gas”: 125
	}

Table C.4: GetMethodDetail API.

API Name	GetMethodDetail
Method	GET
URI	/get-method-detail
Description	API for checking the error distribution of method in smart contract
Request params	1. address (string) 2. method (string) 3. evitar (string)

Response body	{
	“evitar”: “evitar_1”,
	“contract_address” : ”0x3D490ff9e5A7FF...”,
	“methods”: “0xc12fba87”,
	“method_with_high_failure_rated”: true,
	“successful”: 47,
	“revert”: 30,
	“out_of_gas”: 23,
	“invalid_opcode”: 67,
	“invalid_jump”: 35,
	“stack_overflow”: 0,
	“stack_underflow”: 0
	}

Appendix IV

LIST OF PUBLICATIONS

D.1 IEEE Software

1. Jumnongsaksub, S., & Sripanidkulchai, K. (2020). Reducing Smart Contract Runtime Errors on Ethereum. In IEEE Software (Vol. 37, Issue 5, pp. 55–59). https://doi.org/10.1109/MS.2020.2993882

Biography

Siwapol Jumnongsaksub was born in Bangkok on December, 1996. He is a graduate student in the Computer Engineering Department, Chulalongkorn University, Thailand. His research interests include blockchains, smart contracts optimization, and networking. Jumnongsaksub received a B.Eng. in computer engineering from Chulalongkorn University in 2019.