Hybrid Analysis of Smart Contracts and Malicious Behaviors in Ethereum

Xiaoqi Li

PhD

The Hong Kong Polytechnic University

2021

The Hong Kong Polytechnic University
Department of Computing

Hybrid Analysis of Smart Contracts and Malicious Behaviors in Ethereum

Xiaoqi Li

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

November 2020

Abstract

Since its inception, blockchain technology has shown promising application prospects from cryptocurrency to a variety of forms, such as medicine, economics, cloud computing, and so on. As the program deployed and executed in blockchain, smart contract is the core technology in the 2.0 era of blockchain. Through developing smart contracts, developers can realize rich logic and greatly expand the capabilities of blockchain system. As the most popular blockchain system that supports smart contract, Ethereum can complete one million transactions per day.

Since blockchain is one of the core technology in FinTech (Financial Technology) industry, users are very concerned about its security. Some security vulnerabilities and attacks have been recently reported. Note that smart contracts with security vulnerabilities may lead to financial losses. For instance, in June 2016, the criminals attacked the smart contract `DAO` by exploiting a recursive calling vulnerability, and stole around 60 million dollars. In this thesis, we conduct systematic examination on security risks to popular blockchain systems. We survey the real attacks on popular blockchain systems and analyze the vulnerabilities exploited in these cases. Furthermore, we summarize practical academic achievements for enhancing the security of blockchain, and suggest a few future directions in this area.

More than eight million smart contracts have already been deployed in Ethereum, while only less than 1% are open-source. Unfortunately, facing the bytecodes of deployed smart contracts, it is difficult to quickly and comprehensively understand their details. In this thesis, we propose and implement a system named STAN, which can analyze the runtime bytecodes of smart contract and automatically describe its interfaces in natural language, enabling users to quickly and thoroughly understand closed-source contracts. Given the address of target contract, STAN can automatically acquire its runtime bytecodes and describe every interface from four aspects. The functionality description summarizes the interface's functionality, and usage description tells the user how to call this interface. The behavior description describes message-call related behaviors within the interface, and payment description describes whether the interface can receive ETH. We analyze bytecodes through symbolic execution and generate readable descriptions following standard workflow of NLG (Natural Language Generation) system. Furthermore, we statically analyze contract source codes to evaluate descriptions' adequacy and accuracy. We also evaluate descriptions' readability through questionnaires and statistical methods.

Ethereum has two kinds of accounts: EOA (Externally Owned Account) and contract account. However, not all accounts should be kept. We regard the worthless accounts that deserve to be removed without affecting the normal operations of users and other accounts as erasable accounts. Erasable accounts not only waste system resources and affect the efficiency of blockchain, but also easily waste users' money. In this thesis, we design and implement a novel tool named GLASER to discover erasable accounts by analyzing the StateDB of Ethereum. In detail, it leverages program analysis techniques to discover contract accounts with worthless runtime bytecodes, and employs state field and transaction analysis to discover EOAs that no one owns their private keys. The accounts discovered by GLASER are worthless and deserve to be removed without affecting the normal operations of other accounts/users. Applying GLASER to all Ethereum accounts, we discovered 508,482 erasable accounts, and more than 99.9% of them are still stored in Ethereum. These erasable accounts have wasted users more than 106 million dollars and can be removed through executing `SELFDESTRUCT` operation in their runtime bytecodes by users, or removed forcibly by

Ethereum officials. GLASER also characterizes erasable accounts through call graph and creation graph analysis.

Gas is the execution fee for running smart contracts in Ethereum. However, we find that under-optimized smart contracts cost more gas than necessary, and therefore the miners or users will be overcharged. In this thesis, we identify seven gas-costly patterns and divide them into two categories: useless-code related patterns, and loop-related patterns. Furthermore, we propose and develop GASPER, a new tool for discovering gas-costly patterns in bytecode automatically. GASPER leverages symbolic execution and it can locate three representative patterns. By applying GASPER to analyze deployed smart contracts, we find that more than 80% smart contracts suffer from these three patterns, respectively. There are already more than 296 thousand kinds of cryptocurrencies built on Ethereum. However, not all cryptocurrencies can be controlled by users. For example, some money is permanently locked in wallets' accounts due to attacks. In this thesis, we conduct the systematic investigation on locked cryptocurrencies in Ethereum. In particular, we define four categories of accounts with locked cryptocurrencies and develop a novel tool named CLUE to discover them. Results show that there are more than one billion dollars value of cryptocurrencies locked in Ethereum. We also analyze the reasons (i.e., attacks/behaviors) why cryptocurrencies are locked. Because the locked cryptocurrencies can never be controlled by users, avoid interacting with the accounts discovered by CLUE and repeating the same mistakes again can help users to save money.

Keywords: Smart Contract, Ethereum, Program Analysis, Cryptocurrency, Malicious Behavior.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since its inception, blockchain technology has shown promising application prospects from cryptocurrency to a variety of forms, such as medicine, economics, cloud computing, and so on. As the most popular blockchain system that supports smart contract, Ethereum can complete one million transactions per day. Since blockchain is one of the core technology in FinTech industry, users are very concerned about its security. Some security vulnerabilities and attacks have been recently reported. Loi et al. discover that 8,833 out of 19,366 existing Ethereum contracts are vulnerable [134]. Note that smart contracts with security vulnerabilities may lead to financial losses. For instance, the criminals attacked the smart contract `DAO` [11] by exploiting a recursive calling vulnerability, and stole around 60 million dollars. As another example, the criminals exploited transaction mutability in Bitcoin to attack `MtGox`, the largest Bitcoin trading platform. It caused the collapse of `MtGox`, with a value of 450 million dollars Bitcoin stolen [3]. In Chapter 2, we systematically survey security issues for blockchain systems.

More than eight million smart contracts have already been deployed in Ethereum, while only less than 1% are open-source. Unfortunately, facing the bytecodes of deployed smart contracts, it is difficult to quickly and comprehensively understand their details [107] [182], which leads to two issues. First, when users encounter a deployed contract, they usually do not know exactly how to use it, because users do not know the interfaces (i.e., external/public functions) of the bytecodes or the specific functionalities of its interfaces; Second, although some contracts are open-source, the blockchain system only stores the runtime bytecodes of them [62]. Usually common users cannot easily comprehend the contracts' sources published on websites (e.g., Etherscan [24]), not to mention the bytecodes of these contracts. In Chapter 3, we describe the runtime bytecodes of smart contracts in natural language.

Being the largest blockchain that supports smart contract, Ethereum has two kinds of accounts: EOA (Externally Owned Account) and contract account [132]. As a permissionless blockchain system, Ethereum allows any user to create many EOAs through their private keys. Deploying a smart contract to Ethereum will produce a contract account that contains the contract's runtime bytecodes. Every node must synchronize blockchain data, which includes blocks and StateDB (State DataBase) [62]. The StateDB stores all the accounts' state information, such as ETH balance, transaction number, runtime bytecodes, and so on [62]. However, not all accounts should be kept. We regard the worthless accounts that deserve to be removed without affecting the normal operations of users and other accounts as erasable accounts. Erasable accounts not only waste system resources and affect the efficiency of blockchain, but also easily waste users' money. We characterize erasable accounts of Ethereum in Chapter 4.

Gas is the execution fee for running smart contracts in Ethereum. The creators and users of smart contracts will be charged certain amount of gas for purchasing the computing resources from miners. The charge of a transaction equals to the multiplication of the gas consumed by executing the transaction and the price of gas (ETH per unit). Moreover, when deploying contracts, the creators will also be charged of gas, the amount of which are related to the size of smart contracts in bytecodes. However, we find that under-optimized smart contracts cost more gas than necessary, and therefore the miners or users will be overcharged. As the most popular blockchain that supports smart contracts, there are many kinds of contract-based cryptocurrencies built in Ethereum. Apart from ETH, which is the native cryptocurrency of Ethereum, more than 296 thousand cryptocurrency contracts are deployed in Ethereum [54]. These cryptocurrencies have high market capitalization. For example, the ETH has a total value of about 20 billion dollars [56], and USDT has a total value of more than four billion dollars [53]. However, not all cryptocurrencies can be controlled by users. We analyze under-optimized smart contracts and locked cryptocurrencies in Chapter 5.

Thesis Contribution

We make the following contributions in this thesis:

1. We conduct systematic examination on the security of blockchain systems. We survey real attacks and analyze the exploited vulnerabilities. Furthermore, we summarize practical academic achievements for enhancing the security of blockchain, and suggest a few future directions in this area.

2. We propose and implement a system named STAN, which can analyze the runtime bytecodes of smart contract and automatically describe its interfaces

in natural language, enabling users to quickly and thoroughly understand closed-source contracts.

3. We design and implement a tool named GLASER to discover erasable accounts by analyzing the StateDB of Ethereum. It leverages program analysis techniques to discover contract accounts with worthless runtime bytecodes, and employs state field and transaction analysis to discover EOAs that no one owns their private keys.

4. We develop a tool named GASPER for discovering gas-inefficient patterns in bytecodes. GASPER leverages symbolic execution and it can locate three representative patterns. We find that more than 80% smart contracts suffer from these three patterns, respectively.

5. We conduct the systematic investigation on locked cryptocurrencies in Ethereum. We define four categories of accounts with locked cryptocurrencies and develop a tool named CLUE to discover them. Results show that there are more than one billion dollars value of cryptocurrencies locked in Ethereum.

Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we introduce the background of this thesis, including the literature review. We also systematically survey security issues for blockchain systems. In Chapter 3, we describe the runtime bytecodes of smart contracts in natural language, enabling users to quickly and thoroughly understand closed-source contracts. In Chapter 4, we characterize erasable accounts in Ethereum, helping to save system resources and users' money. In

Chapter 5, we analyze under-optimized smart contracts and locked cryptocurrencies, which can help users/developers to save money. Finally, Chapter 6 makes a conclusion of this thesis, and indicates insights obtained from this research and our future work.

The primary research outputs emerged from this thesis are as follows:

- Xiaoqi Li, Ting Chen, Xiapu Luo, Chenxu Wang, "CLUE: Towards Discovering Locked Cryptocurrencies in Ethereum", *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC)*, 2021.

- Xiaoqi Li, Ting Chen, Xiapu Luo, Jiangshan Yu, "Characterizing Erasable Accounts in Ethereum", in *Proceedings of The 23rd Information Security Conference (ISC)*, 2020.

- Xiaoqi Li, Ting Chen, Xiapu Luo, Tao Zhang, Le Yu, Zhou Xu, "STAN: Towards Describing Bytecodes of Smart Contract", in *Proceedings of The 20th IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2020.

- Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, Qiaoyan Wen, "A Survey on the Security of Blockchain Systems", in *Future Generation Computer Systems (FGCS)*, 2017.

- Ting Chen, Xiaoqi Li, Xiapu Luo, Xiaosong Zhang, "Under-optimized Smart Contracts Devour Your Money", in *Proceedings of the Early Research Achievements Track at the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2017.

Chapter 2

Literature Review

Since its inception, the blockchain technology has shown promising application prospects. From the initial cryptocurrency to the current smart contract, blockchain has been applied to many fields. Although there are some studies on the security and privacy issues of blockchain, there lacks a systematic examination on the security of blockchain systems. In this chapter, we conduct a systematic study on the security threats to blockchain and survey the corresponding real attacks by examining popular blockchain systems. We also review the security enhancement solutions for blockchain, which could be used in the development of various blockchain systems, and suggest some future directions to stir research efforts into this area.

Since the debut of Bitcoin in 2009, its underlying technique, blockchain, has shown promising application prospects and attracted lots of attentions from academia and industry. Being the first cryptocurrency, Bitcoin was rated as the top performing currency in 2015 [29] and the best performing commodity in 2016 [4], and has about 300,000 confirmed transactions [38] daily in 2020. At the same time, the blockchain technique has been applied to many fields, including medicine [70, 97, 172], economics [77, 116, 117], Internet of things [96, 158, 179], software engineering [91, 146, 168] and so on. The introduction of Turing-complete programming languages to enable users to develop smart contracts running on the blockchain marks the start of blockchain 2.0 era. With the decentralized consensus mechanism of blockchain, smart contracts allow mutually distrusted users to complete data exchange or transaction without the need of any third-party trusted authority. Ethereum is now the most widely used blockchain supporting smart contracts, where there are already more than eight million deployed smart contracts and more than one million transactions happened daily [131].

Since blockchain is one of the core technology in FinTech (Financial Technology) industry, users are very concerned about its security. Some security vulnerabilities and attacks have been recently reported. Loi et al. discover that 8,833 out of 19,366 existing Ethereum contracts are vulnerable [134]. Note that smart contracts with security vulnerabilities may lead to financial losses. For instance, the criminals attacked the smart contract `DAO` [11] by exploiting a recursive calling vulnerability, and stole around 60 million dollars. As another example, the criminals exploited transaction mutability in Bitcoin to attack `MtGox`, the largest Bitcoin trading platform. It caused the collapse of `MtGox`, with a value of 450 million dollars Bitcoin stolen [3].

Although there are some recent studies on the security of blockchain, none of them performs a systematic examination on the risks to blockchain systems, the corresponding real attacks, and the security enhancements. The closest research work to ours is [69] that only focuses on Ethereum smart contracts, rather than popular blockchain systems. From security programming perspective, their work analyzes the security vulnerabilities of Ethereum smart contracts, and provides a taxonomy of common programming pitfalls that may lead to vulnerabilities [69]. Although a series of related attacks on smart contracts are listed in [69], there lacks a discussion on security enhancement. This chapter focuses on the security of blockchain from more comprehensive perspectives. The main contributions of this chapter are as follows:

- We conduct systematic examination on security risks to popular blockchain systems (e.g., Ethereum, Bitcoin).
- We survey the real attacks on popular blockchain systems and analyze the vulnerabilities exploited in these attack cases.
- We summarize practical academic achievements for enhancing the security of blockchain.

The remainder of this chapter is organized as follows. Section 2.1 introduces the main technologies used in blockchain systems. Section 2.2 systematically examines the security risks to blockchain, and Section 2.3 surveys real attacks on blockchain systems. After summarizing the security enhancements to blockchain in Section 2.4, we summarize the chapter in Section 2.5.

2.1 Overview of Blockchain Technologies

This section introduces the main technologies employed in blockchain. We first present the fundamental trust mechanism (i.e., the consensus mechanism) used in blockchain, and then explain the synchronization process between nodes. After that, we introduce the two development stages of blockchain.

Consensus Mechanism

Being a decentralized system, blockchain systems do not need a third-party trusted authority. Instead, to guarantee the reliability and consistency of the data and transactions, blockchain adopts the decentralized consensus mechanism. In the existing blockchain systems, there are four major consensus mechanisms [180]: PoW (Proof of Work), PoS (Proof of Stake), PBFT (Practical Byzantine Fault Tolerance), and DPoS (Delegated Proof of Stake). Other consensus mechanisms, such as PoB (Proof of Bandwidth) [55], PoET (Proof of Elapsed Time) [47], PoA(Proof of Authority) [46] and so on, are also used in some blockchain systems. The two most popular blockchain systems (i.e., Bitcoin and Ethereum) use the PoW mechanism. Ethereum also incorporates the PoA mechanism (i.e., Kovan public test chain [32]), and some other cryptocurrencies also use the PoS mechanism, such as PeerCoin, ShadowCash and so on.

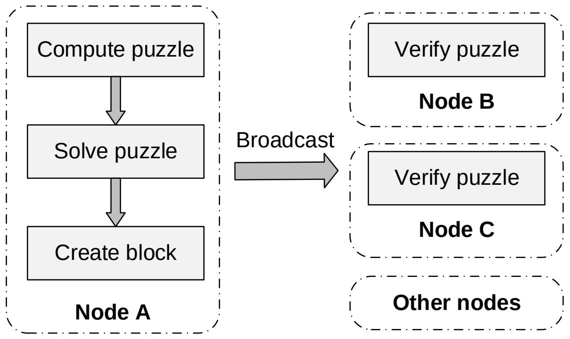


Figure 2.1: PoW consensus mechanism.

PoW mechanism uses the solution of puzzles to prove the credibility of the data. The puzzle is usually a computationally hard but easily verifiable problem. When a node creates a block, it must resolve a PoW puzzle. After the PoW puzzle is resolved, it will be broadcasted to other nodes, so as to achieve the purpose of consensus, as shown in Figure 2.1.

In different blockchain systems, the block structure may vary in detail. Typically in Bitcoin, each block contains `PrevHash`, `nonce`, and `Tx` [136]. In particular, `PrevHash` indicates the hash value of the last generated block, and `Txs` denote the transactions included in this block. The value of `nonce` is obtained by solving the PoW puzzle. A correct `nonce` should satisfy that the hash value shown in Equation 2.1 is less than a target value, which could be adjusted to tune the difficulty of PoW puzzle.

$$SHA256(PrevHash \,||\, Tx1 \,||\, Tx2 \,||\, ... \,||\, nonce) < Target \tag{2.1}$$

PoS mechanism uses the proof of ownership of cryptocurrency to prove the credibility of the data. In PoS-based blockchain, during the process of creating block or transaction, users are required to pay a certain amount of cryptocurrency. If the block or transaction created can eventually be validated, the cryptocurrency will be returned to the original node as a bonus. Otherwise, it will be fined. In the PoW mechanism, it needs a lot of calculation, resulting in a waste of computing power. On the contrary, PoS mechanism can greatly reduce the amount of computation, thereby increasing the throughput of the entire blockchain system.

Block Propagation and Synchronization

In the blockchain, each full node stores the information of all blocks. Being the foundation to building consensus and trust for blockchain, the block propagation mechanisms can be divided into the following categories [106, 123, 124]:

(1) Advertisement-based propagation. This propagation mechanism is originated

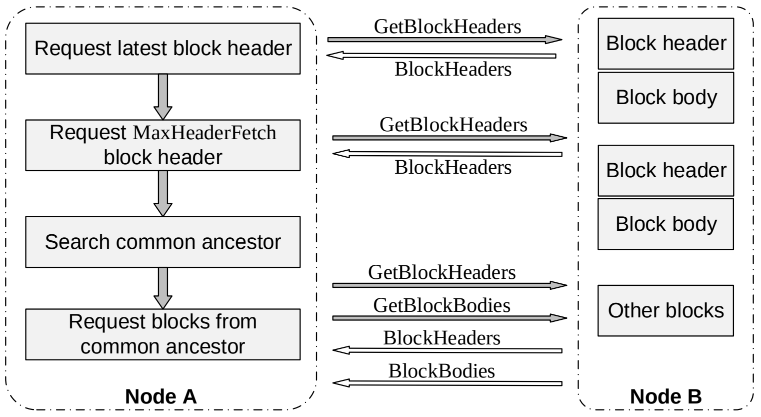


Figure 2.2: Block synchronization process between nodes.

from Bitcoin. When node `A` receives the information of a block, `A` will send an `inv` message (a message type in Bitcoin) to its connected peers. When node `B` receives the `inv` message from `A`, it will do as follows. If node `B` already has the information of this block, it will do nothing. If node `B` does not have the information, it will reply to node `A`. When node `A` receives the reply message from node `B`, node `A` will send the complete information of this block to node `B`.

(2) Sendheaders propagation. This propagation mechanism is an improvement to the advertisement-based propagation mechanism. In the sendheaders propagation mechanism, node `B` will send a `sendheaders` message (a message type in Bitcoin) to node `A`. When node `A` receives the information of a block, it will send the block header information directly to node `B`. Compared with the advertisement-based propagation mechanism, node `A` does not need to send `inv` messages, and hence it speeds up the block propagation.

(3) Unsolicited push propagation. In the unsolicited push mechanism, after one block is mined, the miner will directly broadcast the block to other nodes. In

this propagation mechanism, there is no `inv` message and `sendheaders` message. Compared with the previous two propagation mechanisms, unsolicited push mechanism can further improve the speed of block propagation.

(4) Relay network propagation. This propagation mechanism is an improvement to the unsolicited push mechanism. In this mechanism, all the miners share a transaction pool. Each transaction is replaced by a global ID, which will greatly reduce the broadcasted block size, thereby further reducing the network load and improving the propagation speed.

(5) Push/Advertisement hybrid propagation. This hybrid propagation mechanism is used in Ethereum. We assume that node `A` has n connected peers. In this mechanism, node `A` will push the block to $\sqrt{n}$ peers directly. For the other $n - \sqrt{n}$ connected peers, node `A` will advertise the block hash to them.

Different blockchain systems may use diverse block synchronization processes. In Ethereum, node `A` can request block synchronization from node `B` with more total difficulty. The specific process is as follows (shown in Figure 2.2) [106, 123, 124]:

(1) Node `A` requests the header of the latest block from node `B`. This action is implemented by sending a `GetBlockHeaders` message. Node `B` will reply to node `A` a `BlockHeaders` message that contains the block header requested by `A`.

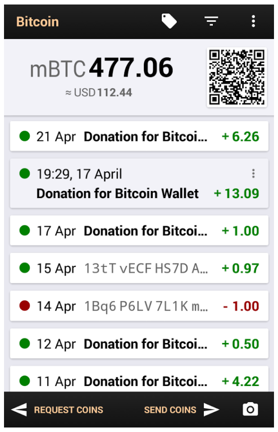
(2) Node `A` requests `MaxHeaderFetch` blocks to find common ancestor from node `B`. The default value of `MaxHeaderFetch` is 256, but the number of block headers sent by node `B` to `A` can be less than this value.

(3) If `A` has not found common ancestor after the above two steps, node `A` will continue to send `GetBlockHeaders` message, requesting one block header each time. Moreover, `A` repeats in binary search to find the common ancestor in its local

blockchain.

(4) After node A discovers a common ancestor, A will request block synchronization from the common ancestor. In this process, A requests `MaxHeaderFetch` blocks per request, but the actual number of nodes sent from B to A can be less than this value.

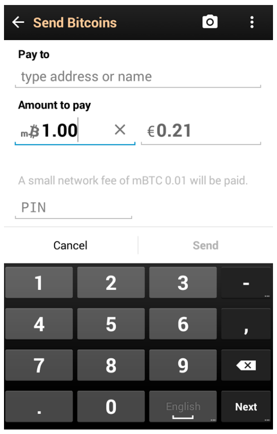
Technology Development



mBTC 477.06
≈ USD 112.44

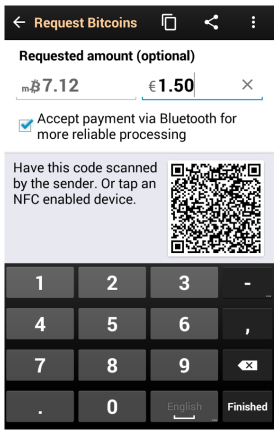
21 Apr Donation for Bitcoi... +6.26
19:29, 17 April Donation for Bitcoin Wallet +13.09
17 Apr Donation for Bitcoi... +1.00
15 Apr 13tT vECF HS7D A... +0.97
14 Apr 1Bq6 P6LV 7L1K m... -1.00
12 Apr Donation for Bitcoi... +0.50
11 Apr Donation for Bitcoi... +4.22

Figure 2.3: Query Bitcoin transaction history.



Pay to
type address or name
Amount to pay
m฿1.00 €0.21
A small network fee of mBTC 0.01 will be paid.
PIN

Figure 2.4: Pay with Bitcoin.



Requested amount (optional)
m฿7.12 €1.50
Accept payment via Bluetooth for more reliable processing
Have this code scanned by the sender. Or tap an NFC enabled device.

Figure 2.5: Collect payments with Bitcoin.

From the birth of the first blockchain system Bitcoin, the blockchain technology has experienced two stages of development: blockchain 1.0 and blockchain 2.0.

In the blockchain 1.0 stage, the blockchain technology is mainly used for cryptocurrency. In addition to Bitcoin, there are many other types of cryptocurrencies, such as Litecoin, Dogecoin and so on. There are currently over 700 types of cryptocurrencies, and the total market capitalizations of them are over 26 billion US$ [12]. The technology stack of cryptocurrency could be divided into two layers: the underlying decentralized ledger layer and protocol layer [159]. Cryptocurrency client, such as

Bitcoin Wallet [7], runs in the protocol layer to conduct transactions, as shown in Figure 2.3 to Figure 2.5. Compared with traditional currency, cryptocurrency has the following characteristics and advantages [60]:

(1) Irreversible and traceable. Transfer and payment operations are irreversible using cryptocurrency. Once the behavior is completed, it is impossible to withdraw. In addition, all user behaviors are traceable, and these behaviors are permanently saved in the blockchain.

(2) Decentralized and anonymous. There is no third-party organization involved in the entire structure of cryptocurrency, nor does it has central management like banks. In addition, all user behaviors are anonymous. Hence, according to the transaction information, we cannot obtain the user's real identity.

(3) Secure and permissionless. The security of the cryptocurrency is ensured by the public key cryptography and the blockchain consensus mechanism, which are hard to be broken by the criminal. Moreover, there is no need to apply for any authority or permission to use cryptocurrency. Users can simply use the cryptocurrency through the relevant clients.

(4) Fast and global. Transactions can be completed in only several minutes using cryptocurrency. Since cryptocurrencies are mostly based on public chains, anyone in the world can use them. Therefore, the user's geographical location has little impact on the transaction speed.

In blockchain 2.0 stage, smart contract is introduced so that developers can create various applications through smart contracts. A smart contract can be considered as a lightweight DApp (decentralized application). Ethereum is a typical system of blockchain 2.0. Each Ethereum node runs an EVM (Ethereum Virtual Machine) that

Table 2.1: Statistics of blockchain systems supporting smart contracts.

System	Contract language	Total TXs	Market Capitalization /M US$
Ethereum	EVM bytecode	23,102,544	8,468
RSK	Solidity	Unknown	N/A
Counterparty	EVM bytecode	12,170,386	15
Stellar	Transaction chains	Unknown	139
Monax	EVM bytecode	Unknown	N/A
Lisk	JavaScript	Unknown	71

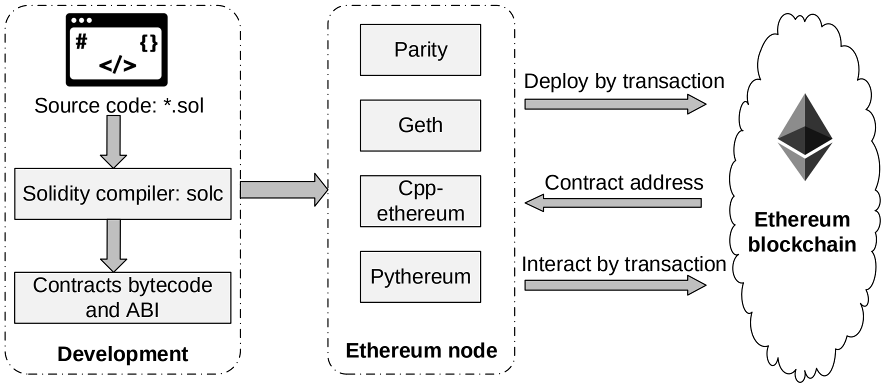


Figure 2.6: The process of smart contract's development, deployment, and interaction.

executes smart contracts. Besides Ethereum, several other blockchain systems also support smart contracts, whose information is listed in Table 2.1 [71]. In Ethereum, developers can use a variety of programming languages to develop smart contracts, such as Solidity (the recommended language), Serpent, and LLL. Since these languages are Turing-complete, smart contracts can achieve rich functions. Figure 2.6 shows the process of smart contracts' development, deployment and interaction. Each deployed smart contract corresponds to a unique address, through which users can interact with the smart contract through transactions by different clients (e.g., Parity, Geth, etc.). Since smart contracts can call each other through messages, developers can develop more feature-rich DApps based on available smart contracts. Compared with the traditional application, a DApp has the following characteristics and advantages [50]:

(1) Autonomy. DApps are developed on the basis of smart contracts, and smart contracts are deployed and run on the blockchain. Hence, DApps can run autonomically without the need of any third party's assistance and participation.

(2) Stable. The bytecodes of smart contracts are stored in the state tree of blockchain. Each full node saves the information of all blocks and stateDB, including the bytecodes of smart contracts. Hence, the failure of some nodes will not affect its operation. This mechanism ensures that DApps can run stably.

(3) Traceable. Since the invocation information of smart contracts is stored in the blockchain as transactions, all the behaviors of DApps are recorded and traceable.

(4) Secure. The public key cryptography and the blockchain consensus mechanism can ensure the security and correct operations of smart contracts, so as to maximize the security of DApps.

2.2 Risks to Blockchain

Table 2.2: Taxonomy of blockchain's risks.

Number	**Risk**	**Cause**	**Range of Influence**
1	51% vulnerability	Consensus mechanism	Blockchain1.0, 2.0
2	Private key security	Public-key encryption scheme	
3	Criminal activity	Cryptocurrency application	
4	Double spending	Transaction verification mechanism	
5	Transaction privacy leakage	Transaction design flaw	
6	Criminal smart contracts	Smart contract application	Blockchain2.0
7	Vulnerabilities in smart contract	Program design flaw	
8	Under-optimized smart contract	Program writing flaw	
9	Under-priced operations	EVM design flaw	

We divide the common blockchain risks into nine categories, as shown in Table 2.2, and detail the causes and possible consequence of each risk. The risks described in Section 3.1 exist in blockchain 1.0 and 2.0, and their causes are mostly related to the blockchain operation mechanism. By contrast, the risks introduced in Section 3.2 are unique to blockchain 2.0, and are usually resulted from the development, deployment, and execution of smart contracts.

2.2.1 Common Risks to Blockchain 1.0 and 2.0

51% Vulnerability

The blockchain relies on the distributed consensus mechanism to establish mutual trust. However, the consensus mechanism itself has 51% vulnerability, which can be exploited by attackers to control the entire blockchain. More precisely, in PoW-based blockchains, if a single miner's hashing power accounts for more than 50% of the total hashing power of the entire blockchain, then the 51% attack may be launched. Hence, the mining power concentrating in a few mining pools may result in the fears of an inadvertent situation, such as a single pool controls more than half of all computing power. In Jan. 2014, after the mining pool `ghash.io` reached 42% of the total Bitcoin computing power, a number of miners voluntarily dropped out of the pool, and `ghash.io` issued a press statement to reassure the Bitcoin community that it would avoid reaching the 51% threshold [6]. In PoS-based blockchains, 51% attack may also occur if the number of coins owned by a single miner is more than 50% of the total blockchain. By launching the 51% attack, an attacker can arbitrarily manipulate and modify the blockchain information. Specifically, an attacker can exploit this vulnerability to conduct the following attacks [2]:

(1) Reverse transactions and initiate double spending attack (the same coins are spent multiple times).

(2) Exclude and modify the ordering of transactions.

(3) Hamper normal mining operations of other miners.

(4) Impede the confirmation operation of normal transactions.

Private Key Security

When using blockchain, the user's private key is regarded as the identity and security credential, which is generated and maintained by the user instead of third-party agencies. For example, when creating a cold storage wallet in Bitcoin blockchain, the user must import his/her private key. Hartwig et al. [139] discover a vulnerability in ECDSA (Elliptic Curve Digital Signature Algorithm) scheme, through which an attacker can recover the user's private key because it does not generate enough randomness during the signature process.

Once the user's private key is lost, it will not be able to be recovered. If the private key is stolen by criminals, the user's blockchain account will face the risk of being tampered by others. Since the blockchain is not dependent on any centralized third-party trusted institutions, if the user's private key is stolen, it is difficult to track the criminal's behaviors and recover the modified blockchain information.

Criminal Activity

Bitcoin users can have multiple Bitcoin addresses, and the address has no relationship with their real life identity. Therefore, Bitcoin has been used in illegal activities. Through some third-party trading platforms that support Bitcoin, users can buy or sell any product. Since this process is anonymous, it is hard to track user behaviors, let alone subject to legal sanctions. Some frequent criminal activities with Bitcoin include:

(1) Ransomware. The criminals often use ransomware for money extortion, and employ Bitcoin as trading currency. In July 2014, a ransomware named `CTB-Locker` [31] spread around the world by disguising itself as mail attachments. If the user clicks the attachment, the ransomware will run in the background of the system and encrypt about 114 types of each file [13]. The victim has to pay the attacker a certain amount of Bitcoin within 96 hours. Otherwise, the encrypted files will not be restored. In May 2017, another ransomware `WannaCry` (also named as `WannaCrypt`) [59] infected about 230,000 victims across 150 countries in two days. It exploited a vulnerability in Windows system to spread, and encrypted users' files to ask for Bitcoin ransom.

Table 2.3: Top 10 categories of items available in `Silk Road`.

Number	**Category**	**Items**	**Percentage**
1	Weed	3338	13.7%
2	Drugs	2194	9.0%
3	Prescription	1784	7.3%
4	Benzos	1193	4.9%
5	Books	955	3.9%
6	Cannabis	877	3.6%
7	Hash	820	3.4%
8	Cocaine	630	2.6%
9	Pills	473	1.9%
10	Blotter (LSD)	440	1.8%

(2) Underground market. Bitcoin is often used as the currency in the underground market. For example, `Silk Road` is an anonymous, international online marketplace that operates as a Tor hidden service and uses Bitcoin as its exchange currency [90]. The top 10 categories of items available in `Silk Road` are listed in Table 2.3 [90]. Most of the items sold in `Silk Road` are drugs, or some other controlled items in the real world. Since international transactions account for a significant proportion in `Silk Road`, Bitcoin makes the transaction in the underground market more convenient, which will cause harm to the social security.

(3) Money laundering. Since Bitcoin has the features like anonymity and network virtual payment and has been adopted by many countries, compared with other currencies, Bitcoin carries the lowest risk of being used for money laundering [58]. Cody et al. propose `Dark Wallet` [14], a Bitcoin application that can make Bitcoin transaction completely stealth and private. `Dark Wallet` can encrypt transaction information and mix the user's valid coins with chaff coins, and hence it can make money laundering much easier.

Double Spending

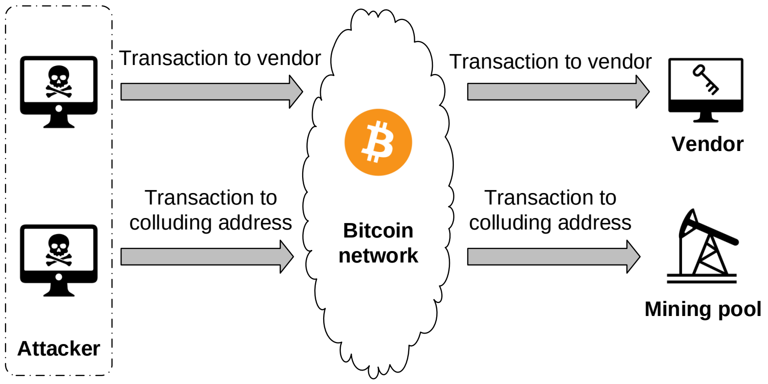


Figure 2.7: Double spending attack model against fast payment in Bitcoin.

Although the consensus mechanism of blockchain can validate transactions, it is still impossible to avoid double spending [122]. Double spending refers to that a consumer uses the same cryptocurrency multiple times for transactions. For example, an attacker could leverage race attack for double spending. This kind of attack is relatively easy to implement in PoW-based blockchains, because the attacker can exploit the intermediate time between two transactions' initiation and confirmation to quickly launch an attack. Before the second transaction is mined to be invalid, the attacker has already got the first transaction's output, resulting in double spending.

Ghassan et al. [121] conduct an analysis of double spending against fast payment in Bitcoin, and propose an attack model, as shown in Figure 2.7. Assuming that an attacker knows the vendor's address before the attack, to perform double spending, the attacker will send two transactions, TX_v and TX_a and choose the same BTCs (cryptocurrency in Bitcoin) as inputs for TX_v and TX_a. TX_v's recipient address is set to the targeted vendor's address, and TX_a's recipient address is set to the colluding address controlled by the attacker. If the following three conditions are met, double spending will be successful: (1) TX_v is added to the wallet of the targeted vendor; (2) TX_a is mined as valid into the blockchain; (3) The attacker gets TX_v's output before the vendor detects misbehavior. If the attack is successful, TX_v will eventually be verified as an invalid transaction, and BTCs are really spent by TX_a. The attacker has received TX_v's output, which is the vendor's normal service. Since TX_a's recipient address is controlled by the attacker, these BTCs are still owned by herself. In this double spending model, the attacker enjoys the service without paying any BTC.

Transaction Privacy Leakage

Since the users' behaviors in the blockchain are traceable, the blockchain systems take measures to protect the transaction privacy of users. In the Bitcoin and Zcash, they use one-time accounts to store the received cryptocurrency. Moreover, the user needs to assign a private key to each transaction. In this way, the attacker cannot infer whether the cryptocurrency in different transactions is received by the same user.

In Monero, users can include some chaff coins (called "mixins") when they initiate a transaction so that the attacker cannot infer the linkage of actual coins spent by the transaction.

Table 2.4: Linkability analysis of Monera transaction inputs with mixins.

	Not deducible	**Deducible**	**In total**
Using newest TXO	15.07%	4.60%	19.67%
Not using newest TXO	22.61%	57.72%	80.33%
In total	37.68%	62.32%	100%

Unfortunately, the privacy protection measures in blockchain are not very robust. Andrews et al. [141] empirically evaluate two linkability weaknesses in Monero's mixin sampling strategy, and discover that 66.09% of all transactions do not contain any mixins. 0-mixin transaction will lead to the privacy leakage of its sender. Since users may use the outputs of 0-mixin transaction as mixins, these mixins will be deducible. Moreover, they study the sampling method of mixins and find that the selection of mixins is not really random. Newer TXOs (transaction outputs) tend to be used more frequently. They further discover that 62.32% of transaction inputs with mixins are deducible, as shown in Table 2.4 [141]. By exploiting these weaknesses in Monero, they can infer the actual transaction inputs with 80% accuracy.

2.2.2 Specific Risks to Blockchain 2.0

Criminal Smart Contracts

Criminals can leverage smart contracts for a variety of malicious activities, which may pose a threat to our daily life. CSCs (Criminal Smart Contracts) can facilitate the leakage of confidential information, theft of cryptographic keys, and various real-world crimes (e.g., murder, arson, terrorism, etc.) [119]. Juels et al. propose an example of password theft CSC `PwdTheft`, whose process is shown in Figure 2.8 [119].

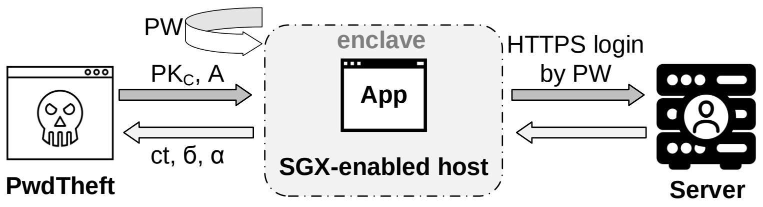


Figure 2.8: Execution procedure of `PwdTheft` using SGX-enabled platform.

`PwdTheft` can be exploited for a fair exchange between contractor `C` and perpetrator `P`. `C` will pay a reward to `P` if and only if `P` gives a valid password to `C`. The entire transaction process can be done without any third party trusted agencies involved. Since the smart contract deployed in blockchain cannot access network directly [176], in the actual work process of `PwdTheft`, it is combined with trusted hardware technology, such as Intel SGX (Software Guard eXtension), to prove the validity of the password through HTTPS (Hypertext Transfer Protocol Secure). SGX will create a trusted execution environment named `enclave`, which can protect the application from being attacked by others. Any privileged or unprivileged software cannot access the runtime environment of `enclave`. Furthermore, SGX can produce `quote`, a digitally signed attestation. `Quote` can get the hash value of the application run in `enclave` environment. Meanwhile, `quote` can access the relevant data during runtime of the application. The whole password exchange process is divided into three steps:

(1) `PwdTheft` provides (pk_C, A), pk_C is the public key of `C`, and `A` is the target account for stealing.

(2) The application that runs in SGX, using the `PW` provided by `P`, logs on to the server account `A` by establishing an HTTPS connection.

(3) If the preceding steps are successful, the data `ct`, σ and α will be transmitted to `PwdTheft`. `ct` $= enc_{pk_C}[PW]$ and $\sigma = Sig_{sk_{app}}[ct]$. sk_{app} is the signature private key of the application. α is a `quote` that runs on `P`'s SGX-enabled host.

After `PwdTheft` receives `ct`, σ and α, `C` can decrypt them to verify the data, and then decide whether a reward should be paid to `P`. In this process, in order to prevent `P` from changing the password maliciously after the data transmission to `PwdTheft`, they can add a timestamp in the data. In addition, `PwdTheft` can be easily extended for conducting other malicious activities. For example, criminals can leverage CSCs to make 0-day vulnerability transactions, which are critical cyber-weaponry [119].

Vulnerabilities in Smart Contract

Table 2.5: Taxonomy of vulnerabilities in smart contract.

Number	**Vulnerability**	**Cause**	**Level**
1	Call to the unknown	The called function does not exist	Contract source code
2	Out-of-gas send	Fallback of the callee is executed	
3	Exception disorder	Irregularity in exception handling	
4	Type casts	Type-check error in contract execution	
5	Reentrancy vulnerability	Function is re-entered before termination	
6	Field disclosure	Private value is published by the miner	
7	Immutable bug	Alter a contract after deployment	EVM bytecode
8	ETH lost	Send ETH to an orphan address	
9	Stack overflow	The number of values in stack exceeds 1024	
10	Unpredictable state	State of the contract is changed before invoking	Blockchain mechanism
11	Randomness bug	Seed is biased by malicious miner	
12	Timestamp dependence	Timestamp of block is changed by malicious miner	

As programs running in the blockchain, smart contracts may have security vulnerabilities caused by program defects. Nicola et al. [69] conduct a systematic investigation of 12 types of vulnerabilities in smart contract, as shown in Table 2.5. Loi et al. [134] propose a symbolic execution tool called OYENTE to find 4 kinds of potential security bugs. They discover that 8,833 out of 19,366 Ethereum smart contracts are vulnerable. The details of these 4 bugs are as follows:

(1) Transaction-ordering dependence. Valid transactions can change the state of Ethereum blockchain from σ to σ': $\sigma \xrightarrow{T} \sigma'$. In every epoch, each miner proposes their own block to update the blockchain. Since a block may contain multiple transactions, blockchain state σ may change multiple times within an epoch. When a new block contains two transactions T_i and T_j, which invoke the same smart contract, it may trigger this vulnerability. Because the execution of the smart contract is associated with state σ, the execution order of T_i and T_j affects the ultimate state. The order of transactions' execution depends entirely on miners. In this case, TOD (Transaction-Ordering Dependent) contracts are vulnerable.

(2) Timestamp dependence. In the blockchain, every block has a `timestamp`. Some smart contracts' trigger conditions depend on `timestamp`, which is set by the miner according to its local system time. If an attacker can modify it, timestamp-dependent contracts are vulnerable.

(3) Mishandled exceptions. This category of vulnerability may occur when different smart contracts are called from each other. When contract `A` calls contract `B`, if `B` runs abnormally, `B` will stop running and return `false`. In some invocations, contract `A` must explicitly check the return value to verify if the call has been executed properly. If `A` does not correctly check the exception information, it may be vulnerable.

(4) Reentrancy vulnerability. During the invocation of the smart contract, the actual state of the contract account is changed after the call is completed. An attacker can use the intermediate state to conduct repeated calls to the smart contract. If the invoked contract involves ETH transfer, it may result in illegal ETH stealing.

Under-Priced Operations

As mentioned earlier, each operation is set to a specific gas value in Ethereum, which can be queried in the yellow paper [62]. Ethereum sets the gas value based on the execution time, bandwidth, memory occupancy and other parameters. In general, the gas value is proportional to the computing resources consumed by the operation. However, it is difficult to accurately measure the consumption of computing resources of an individual operation, and therefore some gas values are not set properly. For example, some IO-heavy operations' gas values are set too low, and hence these operations can be executed in quantity in one transaction. In this way, an attacker can initiate a DoS (Denial of Service) attack on Ethereum.

Table 2.6: Gas modifications in EIP150.

Number	**Operation**	**Old value**	**EIP150 value**
1	EXTCODESIZE	20	700
2	EXTCODECOPY	20	700
3	BALANCE	20	400
4	SLOAD	50	200
5	CALL	40	700
6	SELFDESTRUCT (does not create account)	0	5,000
7	SELFDESTRUCT (creates an account)	0	25,000

Actually, attackers have exploited the under-priced operation `EXTCODESIZE` to attack Ethereum [20]. When `EXTCODESIZE` is executed, it needs to read state information and then the node will read hard disk. Since the gas value of `EXTCODESIZE` is only 20, the attacker can call it more than 50,000 times in one transaction. This will cause the user to consume a lot of computing resources, and block synchronization will be significantly slower compared with the normal situation. As another example, some attackers exploited the under-priced operation `SELFDESTRUCT` to launch DoS attacks [15]. They exploited `SELFDESTRUCT` to create about 19 million empty accounts, which need to be stored in the state tree. This attack caused a waste of hard disk resources. At the same time, the node information synchronization and transaction processing speed are significantly decreased.

In order to solve the security problem caused by under-priced operations, the gas values of seven IO-heavy operations are modified in EIP (Ethereum Improvement Proposal) 150 [34], as shown in Table 2.6. Note that EIP150 has already been implemented in the Ethereum public chain by hard fork, and the new gas table parameters are used after No.2463000 block.

2.3 Blockchain Attack Cases

In this section, we survey real attacks on blockchain systems, and analyze the vulnerabilities exploited in these attacks.

Selfish Mining Attack

The selfish mining attack is conducted by attackers (i.e., selfish miners) for the purpose of obtaining undue rewards or wasting the computing power of honest miners [152]. The attacker holds discovered blocks privately and then attempts to fork a private chain [99]. Afterwards, selfish miners would mine on this private chain, and try to maintain a longer private branch than the public branch because they privately hold more newly discovered blocks. In the meanwhile, honest miners continue mining on the public chain. New blocks mined by the attacker would be revealed when the public branch approaches the length of private branch, such that the honest miners end up wasting computing power and gaining no reward, because selfish miners publish their new blocks just before honest miners. As a result, the selfish miners gain a competitive advantage, and honest miners would be incentivized to join the branch maintained by selfish miners. Through a further consolidation of mining power into the attacker's favor, this attack undermines the decentralization nature of blockchain.

Ittay et al. [99] propose an attack strategy named SELFISH-MINE, which can force the honest miners to perform wasted computations on the stale public branch. In the initial circumstance of SELFISH-MINE, the length of the public chain and private chain are the same. The SELFISH-MINE involves the following three scenarios:

(1) The public chain is longer than the private chain. Since the computing power of selfish miners may be less than that of the honest miners, selfish miners will update the private chain according to the public chain, and in this scenario, selfish miners cannot gain any reward.

(2) Selfish miners and honest miners almost simultaneously find the first new block. In this scenario, selfish miners will publish the newly discovered block, and there will be two concurrently forks of the same length. Honest miners will mine in either of the two branches, while selfish miners will continue to mine on the private chain. If selfish miners firstly find the second new block, they will publish this block immediately. At this point, selfish miners will gain two blocks' rewards at the same time. Because the private chain is longer than the public chain, the private chain will be the ultimate valid branch. If honest miners firstly find the second new block and this block is written to the private chain, selfish miners will gain the first new block' rewards, and honest miners will gain the second new block' rewards. Otherwise, if this block is written to the public block, honest miners will gain these two new blocks' rewards, and selfish miners will not gain any reward.

(3) After selfish miners find the first new block, they also find the second new

block. In this scenario, selfish miners will hold these two new blocks privately, and they continue to mine new blocks on the private chain. When the first new block is found by honest miners, selfish miners will publish its own first new block. When honest miners find the second new block, the selfish miners will immediately publish its own second new block. Then selfish miners will follow this response in turn, until the length of the public chain is only 1 greater than the private chain, after which the selfish miners will publish its last new block before honest miners find this block. At this point, the private chain will be considered valid, and consequently selfish miners will gain the rewards of all new blocks.

DAO Attack

Table 2.7: Some other attacks that exploit smart contracts' vulnerabilities.

Number	**Attack case**	**Related vulnerabilities**
1	King of the ETH throne	Out-of-gas send Exception disorder
2	Multi-player games	Field disclosure
3	Rubixi attack	Immutable bug
4	GovernMental attack	Immutable bug Stack overflow Unpredictable state Timestamp dependence
5	Dynamic libraries attack	Unpredictable state

The `DAO` is a smart contract deployed in Ethereum on 28th May of 2016, which implements a crowd-funding platform. The `DAO` contract was attacked only after it has been deployed for 20 days. Before the attack happened, `DAO` has already raised 150 million US$, which is the biggest crowdfund ever. The attacker stole about 60 million US$.

The attacker exploited the reentrancy vulnerability in this case. Firstly, the attacker publishes a malicious smart contract, which includes a `withdraw()` function `call` to `DAO` in its callback function. The `withdraw()` will send ETH to the callee, which is also in the form of `call`. Therefore, it will invoke the callback function of the malicious smart contract again. In this way, the attacker is able to steal all the ETH from `DAO`. There are some other cases that exploit smart contracts' vulnerabilities (described in Section 2.2.2), which are listed in Table 2.7 [69].

BGP Hijacking Attack

BGP (Border Gateway Protocol) is a de-facto routing protocol and regulates how IP packets are forwarded to their destination. To intercept the network traffic of blockchain, attackers either leverage or manipulate BGP routing. BGP hijacking typically requires the control of network operators, which could potentially be exploited to delay network messages. Maria et al. [68] comprehensively analyze the impact of routing attacks, including both node-level and network-level attacks, on Bitcoin, and show that the number of the successfully to-be-hijacked Internet prefixes depends on the distribution of mining power. Because of the high centralization of some Bitcoin mining pools, if they are attacked by BGP hijacking, it will have a significant effect. The attackers can effectively split the Bitcoin network, or delay the speed of block propagation.

Attackers conduct BGP hijacking to intercept Bitcoin miners' connections to a mining pool server, as analyzed by Dell SecureWorks [5]. By rerouting traffic to a mining pool controlled by the attacker, it was possible to steal cryptocurrency from the victim. This attack collected an estimated 83,000 US$ of cryptocurrency over a two month period. Since the BGP security extensions are not widely deployed, network operators have to rely on monitoring systems, which would report rogue announcements, such as BGPMon [169]. However, even if an attack is detected, solving a hijacking still cost hours as it is a human-driven process consisting of altering configuration or disconnecting the attacker. For example, YouTube ever took about three hours to resolve a hijacking of its prefixes by a Pakistani ISP (Internet Service Provider) [41].

Eclipse Attack

Table 2.8: Some other attacks that may be caused by the eclipse attack.

Number	Attack	Harm
1	Engineering block races	Wasting mining power on orphan blocks
2	Splitting mining power	51% vulnerability may be triggered
3	Selfish mining	Attacker can gain more than normal mining rewards
4	0-confirmation double spend	The vendor would not get rewards for its service
5	N-confirmation double spend	

The eclipse attack allows an attacker to monopolize all of the victim's incoming and outgoing connections, which isolates the victim from the other peers in the network [151]. Then, the attacker can filter the victim's view of the blockchain, or let the victim cost unnecessary computing power on obsolete views of the blockchain. Furthermore, the attacker is able to leverage the victim's computing power to conduct its own malicious acts. Ethan et al. [111] consider two types of eclipse attack on Bitcoin's peer-to-peer network, namely botnet attack and infrastructure attack. The botnet attack is launched by bots with diverse IP address ranges. The infrastructure attack models the threat from an ISP, company or nation-state that has contiguous IP addresses. The Bitcoin network might suffer from disruption and a victim's view of the blockchain will be filtered due to the eclipse attack. Additionally, the eclipse attack is a useful basis for other attacks, as shown in Table 2.8 [111].

Liveness Attack

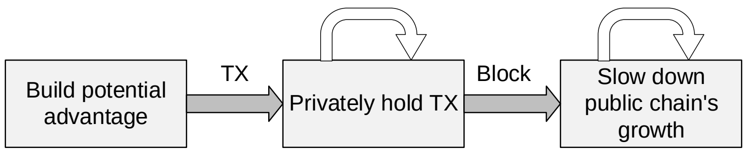


Figure 2.9: Overview of the liveness attack process.

Aggelos et al. [125] propose the liveness attack, which is able to delay as much as possible the confirmation time of a target transaction. They also present two instantiations of such attack on Bitcoin and Ethereum. Liveness attack consists of three phases, namely attack preparation phase, transaction denial phase, and blockchain retarder phase (shown in Figure 2.9):

(1) Attack preparation phase. Just like selfish mining attack, an attacker builds advantage over honest miners in some way before the target transaction TX is broadcasted to the public chain. The attacker builds the private chain, which is longer than the public chain.

(2) Transaction denial phase. The attacker privately holds the block that contains TX, in order to prevent TX from being written into the public chain.

(3) Blockchain retarder phase. In the growth process of the public chain, TX will no longer be able to be privately held in a certain time. In this case, the attacker will publish the block that contains TX. In some blockchain systems, when the depth of the block that contains TX is greater than a constant, TX will be regarded valid. Therefore, the attacker will continue building private chain in order to build an advantage over the public chain. After that, the attacker will publish her privately held blocks into public chain in proper time to slow down the growth rate of public

chain. The liveness attack will end when TX is verified as valid in the public chain.

Balance Attack

Christopher et al. [143] propose the balance attack against PoW-based blockchain, which allows a low-mining-power attacker to momently disrupt communications between subgroups with similar mining power. They abstract blockchain into a DAG (Directed Acyclic Graph) tree, in which DAG $= < B, P >$. B are the nodes indicating blocks' information, and they are connected through directed edges P. After introducing a delay between correct subgroups of equivalent mining power, the attacker issues transactions in one subgroup (called “transaction subgroup”) and mines blocks in another subgroup (called “block subgroup”), to guarantee that the tree of block subgroup outweighs the tree of transaction subgroup. Even though the transactions are committed, the attacker is able to outweigh the tree containing this transaction and rewrite blocks with high probability.

The balance attack inherently violates the persistence of the main branch prefix and allows double spending. The attacker needs to identify the merchant-involved subgroup and create transactions to purchase goods from those merchants. Thereafter, the attacker issues transactions to this subgroup and propagates the mined blocks to the rest nodes of the group. As long as the merchant ships goods, the attacker stops delaying messages. With a high probability that the DAG tree seen by the merchant is outweighed by another tree, the attacker could successfully reissue another transaction using exactly the same coins. Balance attack proves that PoW-based blockchain is block oblivious. That is, when writing a transaction into the main chain, there is a certain probability that the attacker can override or delete the block containing this transaction. In the related experiment, the authors configure an Ethereum private chain with equivalent parameters of R3 consortium [48], and showed that they can successfully carry out the balance attack, which only needs to control about 5% of total computing power.

2.4 Blockchain Security Enhancements

In this section, we summarize security enhancements to blockchain systems, which can be used in the development of blockchain systems.

SmartPool

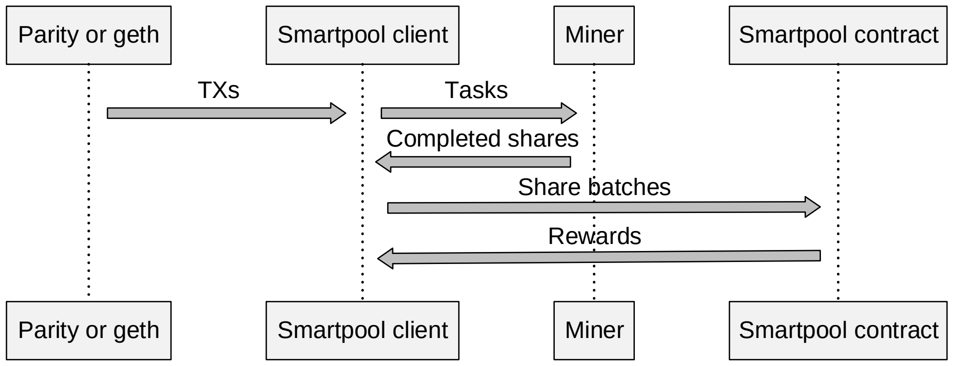


Figure 2.10: Overview of SmartPool's execution process.

As described in Section 2.2.1, there already has mining pool with more than 40% of total computing power of blockchain. This poses a serious threat to the decentralization nature, making blockchain vulnerable to several kinds of attacks. Loi et al. [136] propose a novel mining pool system named SmartPool, whose workflow is shown in Figure 2.10. SmartPool gets the transactions from Ethereum node clients (i.e., parity [43] or geth [39]), which contain mining tasks information.

Then, the miner conducts hashing computation based on the tasks and returns the completed shares to the smartpool client. When the number of the completed shares reaches to a certain amount, they will be committed to smartpool contract, which is deployed in Ethereum. The smartpool contract will verify the shares and deliver rewards to the client. Compared with the traditional P2P pool, SMARTPOOL system has the following advantages:

(1) Decentralized. The core of the SMARTPOOL is implemented in the form of smart contract, which is deployed in blockchain. Miners need first connect to Ethereum to mine through the client. Mining pool can rely on Ethereum's consensus mechanism to run. In this way, it ensures decentralization nature of pool miners. The mining pool state is maintained by Ethereum and no longer requires a pool operator.

(2) Efficiency. Miners can send the completed shares to the smartpool contract in batches. Furthermore, miners only need to send part of shares to be verified, not all shares. Hence, SMARTPOOL is more efficient than the P2P pool.

(3) Secure. SMARTPOOL leverages a novel data structure, which can prevent the attacker from resubmitting shares in different batches. Furthermore, the verification method of SMARTPOOL can guarantee that honest miners will gain expected rewards even there exist malicious miners in the pool.

Quantitative Framework

There exist tradeoffs between blockchain's performance and security. Arthur et al. [106] propose a quantitative framework, which is leveraged to analyze PoW-based blockchain's execution performance and security provisions. As shown in Figure 2.11, the framework has two components: blockchain stimulator and security model. The

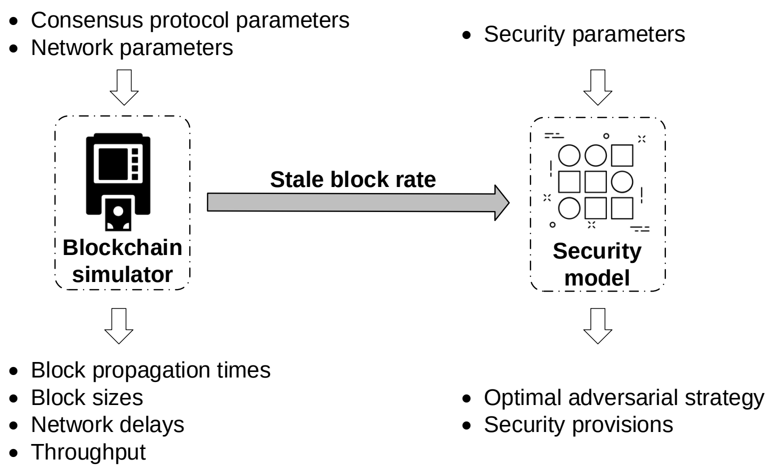


Figure 2.11: Components of quantitative framework.

stimulator mimics blockchain's execution, whose inputs are parameters of consensus protocol and network. Through the simulator's analysis, it can gain performance statistics of the target blockchain, including block propagation times, block sizes, network delays, stale block rate, throughput, etc. The stale block refers to a block that is mined but not written to the public chain. The throughput is the number of transactions that the blockchain can handle per second. Stale block rate will be passed as a parameter to the security model component, which is based on MDP (Markov Decision Processes) for defeating double spending and selfish mining attacks. The framework eventually outputs optimal adversarial strategy against attacks, and facilitates building security provisions for the blockchain.

Oyente

Loi et al. [134] propose Oyente to detect bugs in Ethereum smart contracts. Oyente leverages symbolic execution to analyze the bytecode of smart contracts and it follows the execution model of EVM. Since Ethereum stores the bytecode of smart contracts in its blockchain, OYENTE can be used to detect bugs in deployed contracts.

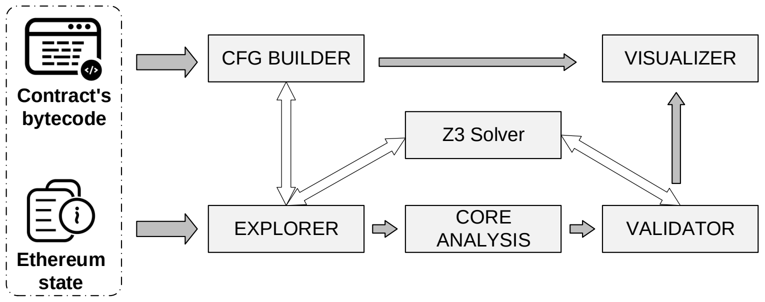


Figure 2.12: Overview of OYENTE's architecture design and execution process.

Figure 2.12 shows OYENTE's architecture and execution process. It takes the smart contract's bytecode and Ethereum global state as inputs. Firstly, based on the bytecode, `CFG BUILDER` will statically build CFG (Control Flow Graph) of smart contract. Then, according to Ethereum state and CFG information, `EXPLORER` conducts simulated execution of smart contract leveraging static symbolic execution. In this process, CFG will be further enriched and improved because some jump targets are not constants; instead, they should be computed during symbolic execution. The `CORE ANALYSIS` module uses the related analysis algorithms to detect four different vulnerabilities (described in Section 2.2.2). The `VALIDATOR` module validates the detected vulnerabilities and vulnerable paths. Confirmed vulnerability and CFG information will finally be output to the `VISUALIZER` module, which can be employed by users to carry out debugging and program analysis. Currently, OYENTE is open source for public use [40].

Hawk

As described in Section 2.2.1, privacy leakage is a serious threat to blockchain. In the era of blockchain 2.0, not only transactions but also contract-related information are public, such as contract's bytecode, invoking parameters, etc.

Ahmed et al. [127] propose Hawk, a novel framework for developing privacy-preserving smart contracts. Leveraging Hawk, developers can write private smart contracts, and it is not necessary for them to use any code encryption or obfuscation techniques. Furthermore, the financial transaction's information will not be explicitly stored in blockchain. When programmers develop Hawk contract, the contract can be divided into two parts: private portion, and public portion. The private data and financial function related codes can be written into the private portion, and codes that do not involve private information can be written into the public portion. The Hawk contract is compiled into three pieces. (1) The program that will be executed in all virtual machines of nodes, just like smart contracts in Ethereum. (2) The program that will only be executed by the users of smart contracts. (3) The program that will be executed by the manager, which is a special trustworthy party in Hawk. The Hawk manager is executed in Intel SGX enclave (described in Section 2.2.1), and it can see the privacy information of the contract but will not disclose it. Hawk can not only protect privacy against the public, but also protect the privacy between different Hawk contracts. If the manager aborts the protocol of Hawk, it will be automatically financially penalized, and the users will gain compensation. Overall, Hawk can largely protect the privacy of users when they are using blockchains.

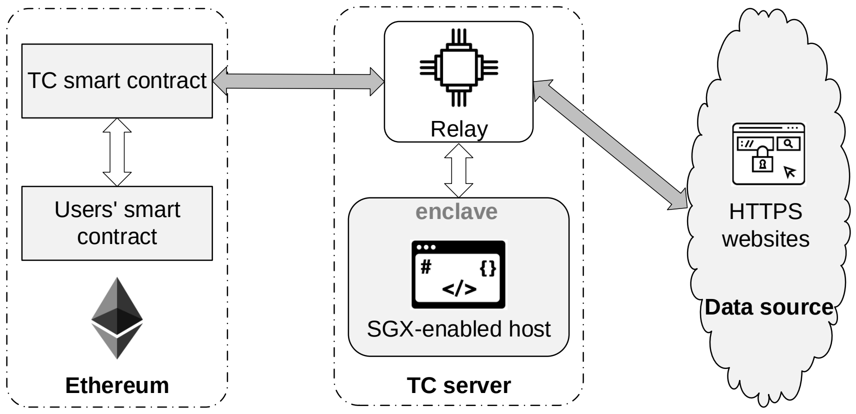


Figure 2.13: Basic architecture of Town Crier system.

Town Crier

Smart contract often needs to interact with off-chain (i.e., external) data source. Zhang et al. [176] propose TC (Town Crier), which is an authenticated data feed system for this data interaction process. Since the smart contract deployed in blockchain cannot access network directly, they cannot get data through HTTPS. TC exactly acts as a bridge between HTTPS-enabled data source and smart contracts. The basic architecture of TC is shown in Figure 2.13. TC contract is the front end of the TC system, which acts as API between users' contracts and TC server. The core program of TC is running in Intel SGX enclave (described in Section 2.2.1). The main function of the TC server is to obtain the data requests from users' contracts, and obtain the data from target HTTPS-enabled websites. Finally, the TC server will return a datagram to the users' contracts in the form of digitally signed blockchain messages.

TC can largely protect the security of the data requesting process. The core modules of TC are respectively running on decentralized Ethereum, SGX-enabled `enclave`, and HTTPS-enabled website. Furthermore, the `enclave` disables the function of network connection to maximize its security. `Relay` module is designed as a network communication hub for smart contracts, SGX `enclave` environment, and data source websites. Therefore, it achieves isolation between network communication and the execution of TC's core program. Even if the `Relay` module is attacked, or the network communication packets are tampered, it will not change the normal function of TC. TC system provides a robust security model for the smart contracts' off-chain data interaction, and it has already been launched online as a public service [57].

2.5 Brief Summary

In this chapter, we focus on the security issues of blockchain technology. By studying the popular blockchain systems (e.g., Ethereum, Bitcoin, Monero, etc.), we conduct a systematic examination on the security risks to blockchain. For each risk or vulnerability, we analyze its causes and possible consequence. Furthermore, we survey the real attacks on the blockchain systems, and analyze the vulnerabilities exploited in these attacks. Finally, we summarize blockchain security enhancements.

In recent period of time, there emerges many new papers related to blockchain security, including smart contract analysis [35] [78] [147] [100] [73] [108] [162] [107] [115], account analysis [103] [126] [126] [81] [129] [167] and cryptocurrency analysis [118] [153] [165] [154] [83] [95] [161]. Due to space limitations in this chapter, we will introduce some of these work in the following chapters.

Chapter 3

Runtime Bytecodes' Description for Smart Contracts

More than eight million smart contracts have been deployed into Ethereum, which is the most popular blockchain that supports smart contract. However, less than 1% of deployed smart contracts are open-source, and it is difficult for users to understand the functionality and internal mechanism of those closed-source contracts. Although a few decompilers for smart contracts have been recently proposed, it is still not easy for users to grasp the semantic information of the contract, not to mention the potential misleading due to decompilation errors. In this chapter, we propose the *first* system named STAN to generate descriptions for the bytecodes of smart contracts to help users comprehend them. In particular, for each interface in a smart contract, STAN can generate four categories of descriptions, including functionality description, usage description, behavior description, and payment description, by leveraging symbolic execution and NLP (Natural Language Processing) techniques. Extensive experiments show that STAN can generate adequate, accurate and readable descriptions for contract's bytecodes, which have practical value for users.

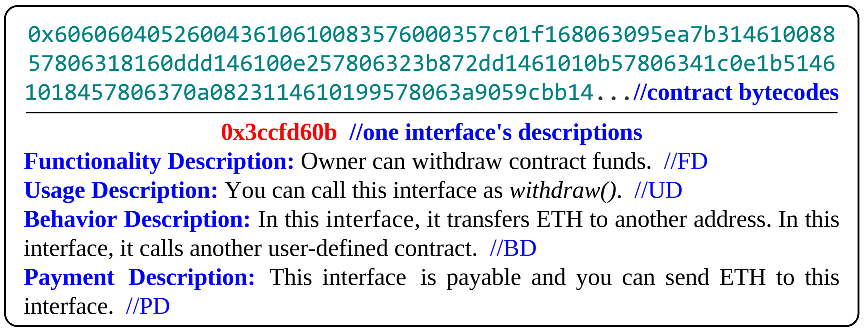
3.1 Overview

Since its inception, blockchain technology has shown promising application prospects from cryptocurrency to a variety of forms, such as medicine [172] [98] and cloud computing [149] [133]. As the program deployed and executed in blockchain, smart contract is the core technology in the 2.0 era of blockchain [180]. Through developing smart contracts, developers can realize rich logic and greatly expand the capabilities of blockchain system. As the most popular blockchain system that supports smart contract, Ethereum can complete one million transactions per day [22]. More than eight million smart contracts have already been deployed in Ethereum, while only less than 1% are open-source [10].

Unfortunately, facing the bytecodes of deployed smart contracts, it is difficult to quickly and comprehensively understand their details [107] [182], which leads to two issues. First, when users encounter a deployed contract, they usually do not know exactly how to use it, because users do not know the interfaces (i.e., external/public functions) of the bytecodes or the specific functionalities of its interfaces; Second, although some contracts are open-source, the blockchain system only stores the runtime bytecodes of them [62]. Usually common users cannot easily comprehend the contracts' sources published on websites (e.g., Etherscan [24]), not to mention the bytecodes of these contracts. Note that all the bytecodes mentioned in this chapter refer to runtime bytecodes.

The root cause of above problems is the lack of tools to comprehensively summarize the functionalities of contract's bytecodes. Although a few decompilers for smart contracts have been recently proposed to turn contracts' bytecodes into user-defined

IR (Intermediate Representation) [107] or Solidity sources [157], it is still not easy for users to grasp the semantic information of the contract, not to mention the potential misleading due to decompilation errors. Some other studies leverage symbolic execution [134] [20], static analysis [160] [34], or formal methods [113] [150] to analyze smart contracts for detecting security issues, whose purposes are different from this chapter.



```
0x6060604052600436106100835760003567c01f168063095ea7b314610088
57806318160ddd146100e257806323b872dd1461010b57806341c0e1b5146
1018457806370a08231146101995780 63a9059cbb14...//contract bytecodes
```

0x3ccfd60b //one interface's descriptions

Functionality Description: Owner can withdraw contract funds. //FD
Usage Description: You can call this interface as *withdraw()*. //UD
Behavior Description: In this interface, it transfers ETH to another address. In this interface, it calls another user-defined contract. //BD
Payment Description: This interface is payable and you can send ETH to this interface. //PD

Figure 3.1: The runtime bytecodes of one closed-source contract and the descriptions for one interface generated through STAN.

In this chapter, we propose and implement a system named STAN (deScribe byTecodes of smArt coNtract), which can analyze the runtime bytecodes of smart contract and automatically describe its interfaces in natural language, enabling users to quickly and thoroughly understand closed-source contracts. One motivating example of STAN's descriptions for a smart contract (address at Mainnet: 0x68854ed29d6feca85242a9b5c00b9e93895a5403) is shown in Figure 3.1. Given the address of target contract, STAN can automatically acquire its runtime bytecodes and describe every interface from four aspects. The functionality description summarizes the interface's functionality, and usage description tells the user how to call this interface. The behavior description describes message-call related behaviors within the interface, and payment description describes whether the interface can receive ETH.

Our Contributions. The main contributions of this chapter are listed as follows:

- To the best of our knowledge, we conduct the *first* research of describing the bytecodes of smart contracts in natural language. For closed-source contract's bytecodes, Stan can generate four categories of descriptions for each interface.
- We leverage program analysis and NLP techniques to describe bytecodes. We analyze bytecodes through symbolic execution and generate readable descriptions following standard workflow of NLG (Natural Language Generation) system.
- We evaluate the generated descriptions from three aspects. We develop a tool named Scans, which statically analyzes contract sources to evaluate descriptions' adequacy and accuracy. We also evaluate descriptions' readability through questionnaires and statistical methods.

The remainder of this chapter is organized as follows. Section 3.2 introduces the technical background, and Section 3.3 details the principle and implementation of Stan. Then Section 3.4 systematically evaluates the generated descriptions. After discussing the limitations and future work in Section 3.5, we summarize the chapter in Section 3.6.

3.2 Background

This section briefly introduces necessary background.

3.2.1 Ethereum

Ethereum has two types of accounts, namely EOA (Externally Owned Account) and contract account [166]. Users can create EOAs and store ETH (native cryptocurrency in Ethereum). Users can send transactions using the private key associated to the EOA address, including ETH transfers and contract calls [84]. The contract account is created by EOA or another contract account. Besides ETH, the contract account contains the bytecodes and storage variables of smart contract.

3.2.2 Smart Contract

In Ethereum, each node runs an EVM (Ethereum Virtual Machine), and the bytecodes of contract are executed in EVM [183]. Smart contract can be developed through several Turing complete languages, such as Solidity (the recommended language), Serpent, and Vyper [183]. Therefore, smart contract can implement complex logics.

Contract Interface: It denotes functions that can be called externally by EOA or other deployed contract. *"external"* or *"public"* functions can be invoked by others. If a contract is open-source in Etherscan [24], the most popular Ethereum block explorer, users can retrieve contract's ABI (Application Binary Interface) to get its interface information.

Contract Invocation: After a smart contract is deployed to Ethereum, its interfaces can be called through transactions [84]. Gas is the basic unit of resource consumption for transactions in Ethereum. Invoking smart contracts through transactions requires a certain amount of gas [81]. When a smart contract is running in EVM, each opcode consumes some gas, whose value is defined in the Yellow Paper [62].

Message-call: There are two kinds of transactions in Ethereum, namely normal transaction (i.e., sent from EOA) and internal transaction (i.e., sent from contract). Through message-call, smart contract can interact with other EOAs or contract accounts, which typically cause internal transactions. One normal transaction may include several internal transactions, and message-call usually comes with the occurrence of sensitive behaviors. For example, through exploiting recursive message-call vulnerability, the criminals stole more than 60 million dollars from smart contract THEDAO [132]. We analyze four different message-call related behaviors in this chapter.

DevDoc: Ethereum NatSpec (Natural Specification) [19] prescribes the writing specifications of DevDoc (Developer Documentation) in the contract's sources. For example, with the annotation field 'details', contract developers can explain the functionality of the interface.

ERCDoc: In EIP (Ethereum Improvement Proposal) [18], there is ERCDoc (ERC Documentation), which prescribes standard interfaces for tokens in Ethereum. For example, ERC20 is the most popular token standard and there already exist more than 238,000 deployed ERC20 tokens [16].

3.2.3 Program Analysis

Loi et al. [134] proposed OYENTE, which uses symbolic execution to detect security bugs in smart contracts. Although STAN uses OYENTE as its symbolic execution engine, our analysis of bytecodes is not related to the security bugs studied in [134]. There are some other symbolic execution engines for detecting vulnerable in smart contracts [37] [35] [78] [128] [142] [144] [147], whose purposes are different from ours.

Sergei et al. [160] proposed SMARTCHECK, a static tool that examines contracts'

Solidity sources to detect security bugs. However, it cannot analyze bytecodes directly. There are many other static analysis tools [100] [74] [108] [162] [163] [110] for detecting different kinds of security issues in smart contracts. Some studies [113] [150] [109] [112] [67] employ formal methods to verify security properties of smart contracts and EVM, whose purposes differ from this chapter.

Matt [157] proposed Porosity, a decompiler for contracts' bytecodes. However, there are still many challenges to generate accurate and readable source codes. Similarly, there are some research [107] [74] [30] [182] [65] [31] decompiling contracts' bytecodes into user-defined intermediate languages, to improve the readability of runtime bytecodes and to facilitate the analysis of smart contracts. Some other studies (e.g., gas optimization [80] [79] [82]) have different purposes. In summary, these studies provide us with valuable inspiration to conduct the first research of describing contracts' bytecodes.

3.3 Stan's Implementation

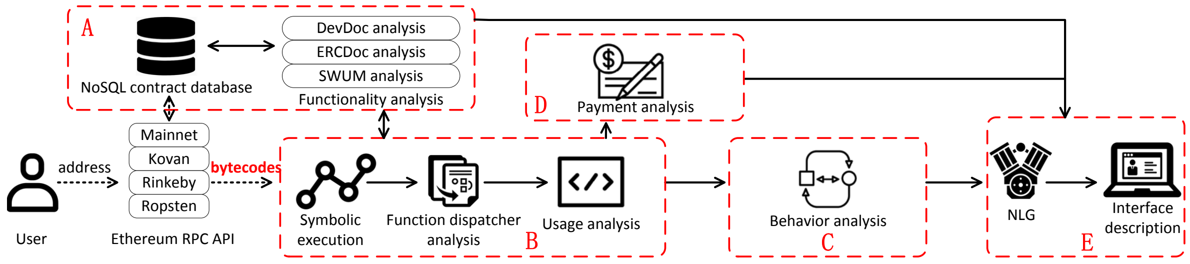


Figure 3.2: Overview of Stan's architecture.

The overview of Stan's architecture is shown in Figure 3.2, which mainly consists of five modules:

(A) Functionality analysis module (Section 3.3.1): Stan conducts contract-oriented analysis through NLP techniques to generate functionality related phrases for interfaces. Note that the functionality of Stan is to describe closed-source contracts' bytecodes, and we analyze open-source contracts and metadata to provide help for bytecodes' analysis through discovering identical bytes signatures.

(B) Usage analysis module (Section 3.3.2): Stan extracts function bytes signatures from function dispatcher and reverse them into corresponding text signatures. From text signature (e.g., $transfer(address, uint256)$), users can know function's name and parameter's configuration, which are used to call the interface.

(C) Behavior analysis module (Section 3.3.3): Stan analyzes external/public functions to generate intermediate information for message-call related behaviors leveraging symbolic execution. Through analyzing opcodes and operands, we recognize four kinds of sensitive message-call behaviors (e.g., ETH transfer, contract deployment, contract call).

(D) Payment analysis module (Section 3.3.4): Stan analyzes external/public functions to generate intermediate information for payment feature through symbolic execution. We construct CFG (Control Flow Graph) to recognize two kinds of payment patterns, indicating whether the interface is payable.

(E) NLG module (Section 3.3.5): Stan generates the final readable interface descriptions leveraging the results of previous four modules. The NLG process follows the standard workflow of NLG system, i.e., document planner, micro-planner, and surface realizer.

3.3.1 Functionality analysis module

DevDoc and ERCDoc analysis

In this section, we analyze DevDoc and ERCDoc to generate phrases that summarize interfaces' functionalities. The result is stored in STAN's database and will be used to facilitate the describing of bytecodes. In the next section, we analyze interfaces without DevDoc or ERCDoc.

Through SCANS' static analysis for 129,737 open-source contracts, there are 5.36% (6,954) sources with DevDoc. We show the process of DevDoc analysis through a function, whose text signature is 'totalSupply()', which is divided into four steps. First, we leverage SCANS to perform static analysis of sources and parse their DevDocs, to extract all the 'details' annotations for functions with signature 'totalSupply()'.

Second, we aggregate the 'details' annotations of functions, whose signatures are the same and appear in different contracts, into a single paragraph. Note that we only intercept the first sentence in each function instance's 'details' annotations, as it is most closer to the goal of describing interfaces' functionalities. In addition, we pre-process the aggregated paragraph. In detail, we remove non-English sentences, identical sentences, meaningless special symbols, etc. After pre-processing, we obtain 53 different sentences, and all of them are written by the developer to describe the functionalities of 'totalSupply()'.

$$W(V_i) = (1 - \overbrace{df}^{Damping\ factor}) + df \times \sum_{V_j \in In(V_i)} \frac{\overbrace{w_{ji}}^{Weight\ of\ E_{ji}}}{\sum_{V_k \in Out(V_j)} w_{jk}} W(V_j) \tag{3.1}$$

where: $In(V_i)$ is the set of vertices that point to V_i,

$Out(V_j)$ is the set of vertices that V_j points to.

Third, we summarize the paragraph T through TextRank Model. TextRank is a ranking model for natural language [140] mainly used to unsupervised keywords extraction for texts. We conduct word segmentation (segmented by spaces), part-of-speech tagging on the paragraph, and filter out stop words. Then we build the keyword graph G = (V, E) through TextRank Model, whose vertice set is composed of word t_i. If two different t_is appear in a window of length k, they have the co-occurrence relationship and there is an edge between the corresponding two vertices with specified weights. V_i's weight is computed using Formula 3.1. We sort all the vertices according to their weights, to get several words with top weight values as keywords. At last, we extract key phrases (i.e., keywords with co-occurrence relationship) as summarization of the paragraph.

$$Similarity(S_i, P_j) = \frac{|\{\overbrace{w_m}^{Words\ in\ sentence\ and\ phrase}|w_m \in S_i \cap w_m \in P_j\}|}{|\{w_n|w_n \in S_i \cup w_n \in P_j\}|} \tag{3.2}$$

Table 3.1: Part of the statistics of ranked sentences in 'details' paragraph for function signature 'totalSupply()'.

ID	**MinHash Jaccard index**	**Sentence**
⋆47⋆	⋆0.183017870949962⋆	⋆'Total supply of tokens.'⋆
36	0.161335751783794	'Returns the total token supply.'
43	0.140755293788616	'Function to access total supply of tokens.'
2	0.139558685183205	'Total Supply.'
1	0.114068411467280	'Retrieves total supply.'
40	0.091591782314505	'Obtain total number of tokens in existence.'

Fourth, to get the significance weight for different sentences in the paragraph, we calculate Jaccard index (shown in Formula 3.2) of each sentence S_i to extracted key phrases P_j through MinHash algorithm [76]. At last, we sort the sentences according to their weight values, as shown in Table 3.1 (ID represents position ordinal of sentence in the 'details' paragraph), and select the highest weighted sentence (marked with $\star$) as the functionality phrase for the function with signature 'totalSupply()'.

Furthermore, we find that token-related function signatures have very high occurrence frequencies. Through SCANS' static analysis for 129,737 open-source contracts, there are 67.56% (87,652) sources with contract names that contain the keyword 'erc'. In other words, approximately 67.56% contracts implement the ERC standard token interfaces. Therefore, we get ERCDocs from EIPs and analyze standard token interfaces, to obtain token-related function signatures and their corresponding functionality phrases.

Table 3.2: Quantity statistics of token interfaces defined in ERCDocs.

Documentation	**Defined interfaces**	**Documentation**	**Defined interfaces**
ERC20	9	ERC918	9
ERC721	17	ERC998	25
ERC777	15	ERC1080	8
ERC827$\star$	$\underline{9}$+3	ERC1132$\star$	$\underline{0}$+9
ERC884$\star$	$\underline{6}$+11	ERC1203$\star$	$\underline{6}$+4
ERC900	10	ERC1410	12

Because the writing structure of ERCDocs is not standardized or unified, it is difficult to parse their content automatically. First, we analyze the ERCDocs manually, extracting function signatures and their corresponding annotations defined in the documents. We have analyzed 12 popular token-related ERCDocs, and their relevant statistics are shown in Table 3.2. $\star$ marks ERCDocs that extend ERC20. For example, ERC827 inherits 9 functions from ERC20 and defines 3 new functions. Note that the ERC1132 is also ERC20's extension; however, it only describes its 9 new functions. For all ERCDocs, we only extract external/public functions. Second, we combine different annotations of the same function signature into one paragraph. Third, we summarize functions' paragraphs through TextRank model separately, to generate functionality phrases for interfaces defined in ERCDocs. Eventually, we generate 115 different function signatures and their corresponding functionality phrases, which will be loaded into STAN's database.

SWUM analysis

In this section, we generate functionality related phrases through SWUM for interfaces without DevDoc or ERCDoc, whose process is divided into three steps.

SWUM (Software Word Usage Model) is used to extract linguistic information from program statements, including words in different parts of speech and the language relationship between them [156]. We use some examples to interpret the process. First, for functions that follow standard naming conventions, we segment their text signatures through specific rules. We analyze three types of naming conventions, i.e., Camel case (e.g., 'isPresaleReady()'), Pascal case (e.g., 'GiveBlockReward()'), and Snake case (e.g., 'claimed_tokens()'). For those functions that do not follow standard naming conventions, we leverage Zipf's law [61] to conduct word segmentation. After word segmentation, we tag the words in part-of-speech to get a set of nouns, verbs, and so on. For example, the function signature 'isPresaleReady()' is segmented into ('is', 'presale', 'ready') and tagged as ('VBZ', 'NP', 'ADJP').

Second, we analyze the linguistic relationship between the segmented words, such as subject-verb, verb-object, passive relations, etc. Leveraging Stanford parser [145], we construct syntax tree of the text signature to analyze its linguistic relationship. We analyze four types of syntax tree, including SINV (inverted declarative sentence), FRAG (fragment), S (simple declarative clause), and NP (noun phrase). The

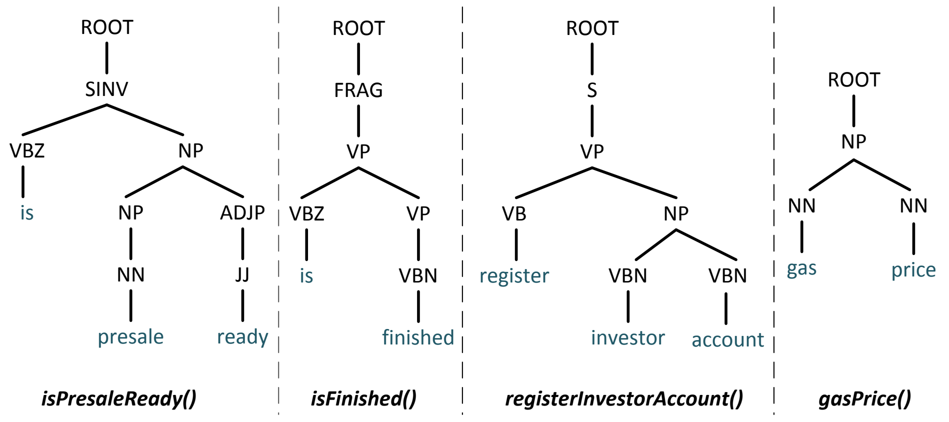


Figure 3.3: Four types and examples of syntax tree for function signature.

structures and examples of these four syntax trees are shown in Figure 3.3. SINV represents inverted declarative sentence; FRAG represents fragment; S represents simple declarative clause; NP represents noun phrase.

Third, the functionality phrase is generated using the analysis results from the previous two steps, and the process is shown in Algorithm 1. In the algorithm of phrase generation, we analyze the structure of syntax tree, then select artificially designed verbs and templates to be assembled as phrases. For example, the function 'isPresaleReady()' is classified as SINV type of syntax tree, and then we depth-first traverse the syntax tree, looking for the noun subject. Afterward, we select verb 'check' and template *'whether the NP VBZ ADJP'*, and generate the functionality phrase for this function as *'Checks whether the presale is ready'*.

Algorithm 1: Phrase generation through syntax tree

```
1.  **Input:** $S_f \leftarrow$ signature of function $f$
2.  $List_f \leftarrow$ WordSegmentation($S_f$) ▷through Camel, Pascal, Snake cases, or Zip'f law
3.  $List_f \leftarrow$ Part-of-speech($List_f$)
4.  $ST_f \leftarrow$ StanfordParser($List_f$) ▷construct syntax tree
5.  Switch(Type($ST_f$)):
6.    Case $SINV$: ▷inverted declarative sentence
7.      $VE \leftarrow$ VerbSelector($ST_f$)
8.      $TP_f \leftarrow$ TemplateSelector($ST_f$)
9.      $P_f \leftarrow$ PhraseConstructor($List_f$, $ST_f$, $VE$, $TP_f$)
10.   Case $FRAG$: ▷fragment
11.     $OB \leftarrow$ ObjectSelector($ST_f$)
12.     $TP_f \leftarrow$ TemplateSelector($ST_f$)
13.     $P_f \leftarrow$ PhraseConstructor($List_f$, $ST_f$, $OB$, $TP_f$)
14.   Case $S$: ▷simple declarative clause (no need to add new elements or select
      templates, because $List_f$ can be fully constructed into a phrase, and the sentence
      structure is fixed)
15.     $P_f \leftarrow$ PhraseConstructor($List_f$, $ST_f$)
16.   Case $NP$: ▷noun phrase (no need to select templates, because the sentence
      structure is fixed)
17.     $VE \leftarrow$ VerbSelector($ST_f$)
18.     $P_f \leftarrow$ PhraseConstructor($List_f$, $ST_f$, $VE$)
19. **Output:** $P_f$
```

Database construction

We have crawled total of 129,737 deployed open-source contracts from Etherscan, whose statistics are shown in Table 3.3 (⋆ marks Testnets). Through DevDoc and ERCDoc analysis, we have generated 12,993 and 115 different function signatures and their corresponding functionality phrases respectively. Leveraging SCANS, we totally extract 2,860,798 function text signatures from 129,737 sources' ABIs. As supplements, we obtain 147,724 from EFSD (Ethereum Function Signature Database) [17], which is a public signature database that anyone can updates. Then we combine the text signatures extracted from ABIs and EFSD, removing duplicates, and use Keccak-256 hash algorithm to calculate bytes signature for each item. Eventually, we obtain 202,995 different text signatures and corresponding bytes signatures, which are used in SWUM analysis and usage analysis (Section 3.3.2). We publish the above analysis results data on `https://figshare.com/articles/dataset/11650734`. To the best of our knowledge, it is the most comprehensive Ethereum function signature public dataset.

Table 3.3: Quantity statistics of deployed open-source smart contracts.

Network name	**Transactions**	**Block depth**	**Open-source contracts**
Mainnet	506,822,407	8,234,086	**50,017**
Kovan⋆	23,994,109	12,485,019	**8,622**
Rinkeby⋆	38,609,478	4,807,975	**19,527**
Ropsten⋆	106,730,113	6,073,746	**51,571**

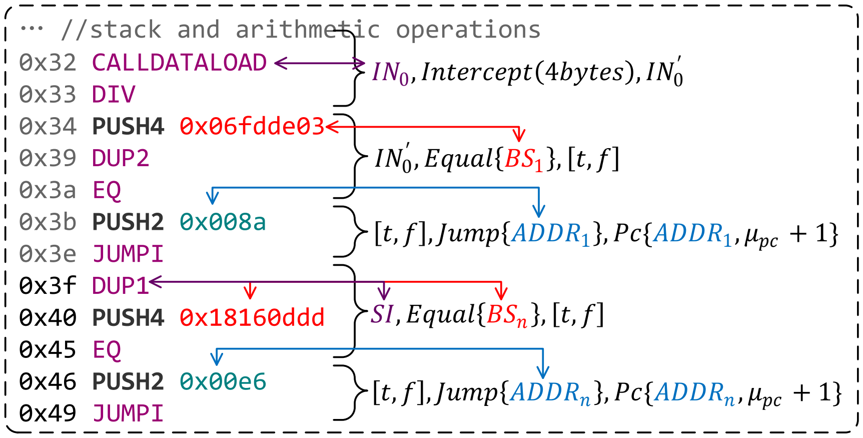
Note that we do not leverage code clone techniques in this chapter because we only analyze open-source contract's function signature, not its function body. STAN can describe bytecodes that do not have corresponding sources. The STAN's database is used to reverse bytes signature and help to generate functionality and usage descriptions. Behavior and payment descriptions' generation does not use the database at all. We also use two different bytecodes datasets, one has corresponding sources and the other one does not have, to fully evaluate STAN in Section 3.4.

We construct contract database for STAN, and import these function-related data to assist in describing bytecodes of closed-source contracts. If one function's bytes signature in bytecodes can be retrieved in the database, we can use its related data to generate functionality and usage descriptions. We use MongoDB [89] to implement the database, and fully import the datasets published in the above URL into database to help describe bytecodes. When the final functionality descriptions are generated for one interface, we set specific priority rules to decide which field is used, which are presented in Section 3.3.5. The results of SWUM analysis are not loaded into database, because STAN directly generates descriptions from function signatures through SWUM if their DevDoc-related or ERCDoc-related phrases cannot be retrieved in database. Note that if one function's bytes signature in bytecodes cannot be retrieved in the database, its functionality and usage descriptions may not be generated properly.

3.3.2 Usage analysis module

In this section, we analyze the runtime bytecodes to recognize external functions' signatures, further to generate intermediate information for usage descriptions.

The usage analysis is divided into three steps. First, leveraging OYENTE [134], which is a symbolic execution engine, we construct CFG of the runtime bytecodes. Second, we recognize function dispatchers in the CFG. The function dispatcher is used to compare the bytes signature encoded in transaction parameter with signatures



```
··· //stack and arithmetic operations
0x32 CALLDATALOAD
0x33 DIV
0x34 PUSH4 0x06fdde03
0x39 DUP2
0x3a EQ
0x3b PUSH2 0x008a
0x3e JUMPI
0x3f DUP1
0x40 PUSH4 0x18160ddd
0x45 EQ
0x46 PUSH2 0x00e6
0x49 JUMPI
```

$IN_0, Intercept(4bytes), IN_0'$
$IN_0', Equal\{BS_1\}, [t, f]$
$[t, f], Jump\{ADDR_1\}, Pc\{ADDR_1, \mu_{pc} + 1\}$
$SI, Equal\{BS_n\}, [t, f]$
$[t, f], Jump\{ADDR_n\}, Pc\{ADDR_n, \mu_{pc} + 1\}$

Figure 3.4: Bytecode snippet of two different types of function dispatcher.

in runtime bytecodes, to decide which function to execute exactly. There exist two different types of function dispatchers in bytecodes, and we use the bytecode snippet of one closed-source contract (Address at Mainnet: 0x50e57ada51fa82b5a3de6ebae3d2 1f88c8d3a672) (shown in Figure 3.4) to interpret their patterns. Opcodes in program counter 0x32 to 0x3e belong to dispatcher type one, and opcodes in counter 0x3f to 0x49 belong to dispatcher type two. For the first type, which is usually located in the opcode block of initial entrance, it reads the first 32 bytes of the transaction's input data as IN_0. After intercepting the first 4 bytes of IN_0 into IN_0', it compares IN_0' with the first bytes signature in bytecodes BS_1. If IN_0' equals BS_1, the program counter will be changed to $ADDR_1$, which is the opcode block corresponding to function BS_1. Otherwise, the program counter will be changed to $\mu_{pc} + 1$. For the second type of function dispatcher, it reads SI, which is the bytes signature of the transaction's target function, from the stack directly. Then it compares SI with one of bytes signature in bytecodes BS_n. If SI equals BS_n, the program counter will be changed to $ADDR_n$. Otherwise, the program counter will be changed to $\mu_{pc} + 1$.

Third, we extract function bytes signatures from function dispatchers (i.e., $(BS_1, 0x06fdde03)$ and $(BS_n, 0x18160ddd)$) and retrieve their corresponding text signatures through the contract database. Note that it may fail to retrieve text signatures, whose adequacy is evaluated in Section 3.4.2. Eventually, the extracted function bytes signatures and their corresponding text signatures (i.e., $(0x06fdde03, name()$ and $(0x18160ddd, totalSupply())$ act as intermediate information to be transferred to the NLG module (in Section 3.3.5) to generate usage descriptions.

3.3.3 Behavior analysis module

Table 3.4: Four different categories of interface behaviors through message-call, and their corresponding opcodes and operands to be analyzed.

Interface behavior	Analyzed opcode	Analyzed operand
ETH transfer⋆	CALL, CALLCODE	P_v
	SELFDESTRUCT	✗
Pre-compiled contract call	CALL	P_a
User-defined contract call⋆	CALL	P_a
	CALLCODE, STATICCALL, DELEGATECALL	✗
Contract deployment⋆	CREATE	✗

In this section, we analyze four kinds of message-call behaviors in interface, further to generate intermediate information for behavior descriptions. The behavior analysis is divided into two steps. First, for every execution path in function body, we record the occurrence of message-call related opcodes and their corresponding operands through symbolic execution. Second, we analyze the recorded information of message-call related opcodes and operands, and summarize it into four different categories of interface behaviors listed in Table 3.4 (⋆ marks the behaviors that cause internal transactions).

For ETH transfer behavior, there are two scenarios. In the first scenario, `CALL` or

CALLCODE is bound to appear during function body execution. Further, we analyze their value field operand P_v, to check whether P_v is a non-zero constant or symbolic value. If P_v is a non-zero constant, it indicates the existence of a fixed amount of ETH transfer. If P_v is a symbolic value, it indicates the existence of a non-fixed amount of ETH transfer, whose specific value is determined by function's input parameter or contract's storage. In the second scenario, SELFDESTRUCT is bound to appear and we do not need to analyze its operand. The occurrence of SELFDESTRUCT indicates that there is ETH transfer during contract's self-destruction. For PRE contract call behavior, CALL is bound to appear, and we analyze its target address field operand P_a, to check whether P_a is a constant value from *0x1* to *0x8*. From version Metropolis [62], Ethereum implements eight different PRE contracts.

For user-defined contract call behavior, there are two scenarios. In the first scenario, CALL is bound to appear during function body execution, and its operand P_a is not any of the addresses of PRE contracts. In the second scenario, CALLCODE or STATICCALL or DELEGATECALL is bound to appear during function body execution. We do not need to analyze their operands in this scenario, and the presence of any of them can prove the existence of user-defined contract call behavior.

For contract deployment behavior, CREATE is bound to appear during function body execution, and we do not need to analyze its operands. The occurrence of CREATE indicates that there is inline assembly or call to its constructor in the function body, to deploy new contracts. At last, the target interface's specific message-call behavior category will act as intermediate information to be transferred to the NLG module to generate behavior descriptions.

3.3.4 Payment analysis module

In this section, we analyze whether the target interface is ETH payable, further to generate intermediate information for payment feature descriptions. When we develop smart contract with Solidity, a function needs to be decorated with modifier *payable* in order to receive ETH through transactions. If users call a non-payable interface with ETH, the transaction execution will fail and waste user's gas.

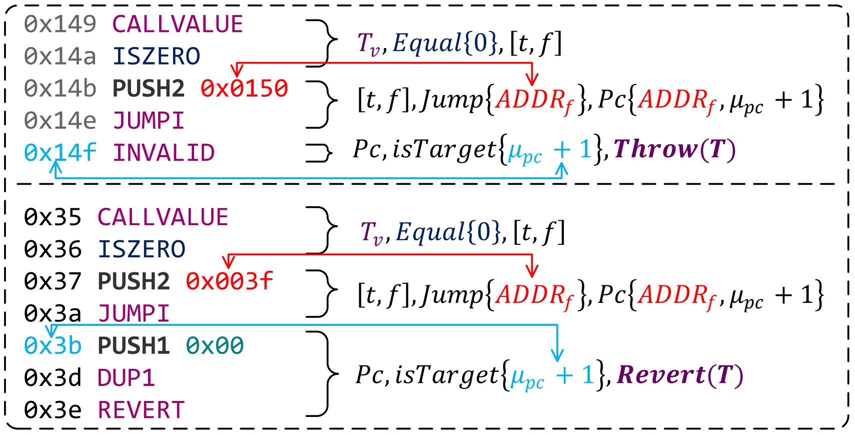


Figure 3.5: Bytecode snippet of two different types of payment operations.

We recognize payment operations' patterns in the CFG of the target function body, to detect whether the interface is non-payable. We use two different bytecode snippets (contract A at Mainnet: 0x6ab6aac6a6f844e322a6c42b3185e1bc4cf56e42, and B at Mainnet: 0xa3ed88f7c9bf7df33b7549bb8c5a889b6049504c) to interpret payment patterns as shown in Figure 3.5. Opcodes in program counter 0x149 to 0x14f (in function $0x66117276$) belong to non-payable type one, and opcodes in counter 0x35 to 0x3e (in function $0x62c06767$) belong to non-payable type two.

It acquires transaction's value field T_v, and determine whether T_v equals 0. If

T_v equals 0, the program counter will be changed to $ADDR_f$, which is the initial execution block's address of behaviors within function. If T_v is not equal to 0, the program counter is changed to $\mu_{pc} + 1$, which is the address of next execution block. In the execution block that started from $\mu_{pc} + 1$, it throws or reverts the transaction due to different compiler SOLC [51] versions. Before SOLC version 0.4.12, it throws the transaction through `INVALID` (*0xfe* in bytes). In this scenario, it rolls back all state changes and consumes the remaining gas. For example, one transaction (Hash at Rinkeby: 0x128907301beff5c56af8234e3c925567352696defffc89e453313e33ae73a c5d) failed with *invalid opcode error*, and it consumed all the 4,707,786 gas from the user. After SOLC version 0.4.12 (including v0.4.12), it reverts the transaction through `REVERT` (*0xfd* in bytes). In this scenario, it rolls back all state changes and returns the remaining gas to the user. For example, one transaction (Hash at Mainnet: 0x037a08b19bc3255e2feca42f6e08294ab8f7daa26e400d998b2a1368159216f2) failed with *inverted error*, and it consumes 22.53% (22,525/100,000) of the gas given by the user. Therefore, in both scenarios, transaction will fail and cause the user's gas wasted. Note that either of our detected pattern's occurrence indicates that the target interface is non-payable.

At last, the result of interfaces' analysis, $(0x66117276, [Nonpayable, True])$ and ($0x62c06767, [Nonpayable, True]$), act as intermediate information to be transfered to the NLG module respectively to generate feature descriptions.

3.3.5 NLG module

In this section, we generate the final readable interface descriptions leveraging the results of previous four analysis modules. The NLG module mainly consists of three steps, which are shown in Algorithm 2.

Algorithm 2: Description generation through NLG module

```
1. Input: $IF_{f,u,b,p}$ ▷intermediate information of functionality, usage, behavior, and payment descriptions
2. Step 1: document planner
3. $WIF \leftarrow$ WeightAssign($IF_{f,u,b,p}$)
4. For $Element$ in $IF_{f,u,b,p}$:
5.   $WIF_{f,u,b,p} \leftarrow$ WeightAssign($Element$)
6. Step 2: micro-planner
7. For $P_{d,s,e}$ in $IF_f$:
8.   $C_{G_{d,s,e}} \leftarrow$ NLGFactory-CreateClause($P_{d,s,e}$)
9. For $Element$ in $IF_{u,b,p}$:
10.   $TP_{u,b,p} \leftarrow$ TemplateSelector(Type($IF_{u,b,p}$))
11.   $C''_{U,B,P} \leftarrow$ NLGFactory-CreateClause($TP_{u,b,p}$,$Element$)
12. $C'_{U,B,P} \leftarrow$ Aggregator($C''_{U,B,P}$)
13. Step 3: surface realizer
14. $C_F \leftarrow$ WeightHighest($C_{G_{d,s,e}}$,$WIF_g$)
15. $C_{U,B,P} \leftarrow$ WeightSort($C'_{U,B,P}$,$WIF_{u,b,p}$)
16. $D_{F,U,B,P} \leftarrow$ NLGFactory-CreateParagraph($C_{F,U,B,P}$)
17. $D_i \leftarrow$ GrammarChecker(WeightSort($D_{F,U,B,P}$,$WIF$))
18. Output: $D_i$
```

The first step is document planner for content determination and document structuring. After importing four categories of intermediate information $IF_{f,u,b,p}$ for the target interface, we give them four different weights to determine the order in which the descriptions appear. Note that specific weight values of WIF are set depending on the degree of IF's importance, which will be used to determine paragraphs' order. For example, we give IF_f the highest weight and functionality description will appear first among the four kinds of descriptions. Similarly, we traverse specific elements in $IF_{f,u,b,p}$ and weight them, which will be used to select elements to generate sentences, and to determine the order of sentences within paragraphs.

The second step is micro-planner for lexicalization and aggregation. Through IF_f, we parse three different kinds of functionality phrases $P_{d,s,e}$, which represent DevDoc-based, SWUM-based, and ERCDoc-based phrases. Leveraging NLGFactory APIs in SimplenLG [105], which is a package used for language generation, we create complete sentences $C_{G_{d,s,e}}$ of functionality descriptions from $P_{d,s,e}$. Then we traverse specific elements in $IF_{u,b,p}$ and select sentence template $TP_{u,b,p}$ according to the category of $IF_{u,b,p}$. Using specific $TP_{u,b,p}$ and $Element$, we create complete sentences for usage, behavior, and payment descriptions $C''_{U,B,P}$. Because there might exist sentences that are highly similar or identical in $C''_{U,B,P}$, we set rules to aggregate these sentences. For example, when there are two ETH transfers in the same IF_b, we describe them only once.

The third step is surface realizer for linguistic and structure realization. For the three kinds of sentences $C_{G_{d,s,e}}$, we select the highest weighted sentence to act as the interface's functionality description. In NLG module's implementation, we set the highest weights for ERCDoc-based sentences because their IF_f are artificially analyzed and extracted from EIPs in Section 3.3.1, which are more accurate than the other two kinds of sentences. For the sentences in $C'_{U,B,P}$, we determine their appearance order according to their weight values in $WIF_{u,b,p}$. Then we create four different paragraphs $D_{F,U,B,P}$ from the four kinds of sentences $C_{F,U,B,P}$, leveraging NLGFactory APIs. After we adjust paragraphs' order of $D_{F,U,B,P}$ according to their weight values in WIF, we check their language grammar through LANGUAGECHECK [33]. At last, D_i is output as the final descriptions for the target interface.

3.4 Evaluation

We conduct a series of experiments to evaluate STAN, which are used for answering the following research questions:

RQ1 Adequacy: How many contracts' bytecodes can be successfully described through STAN?

RQ2 Accuracy: To what extent can STAN accurately describe contracts' bytecodes?

RQ3 Readability: How is the readability of the generated descriptions for users?

3.4.1 Datasets and Experimental Overview

Table 3.5: Quantity statistics of two kinds of contract bytecodes' datasets for evaluation.

DS	Network	Bytecode	Destructed	Identical	Analyzed
1	Mainnet	6,920,465	N/A	6,803,635	116,830
2	Mainnet	50,017	725	1,398	47,894
	Kovan	8,622	79	514	8,029
	Rinkeby	19,527	228	269	19,030
	Ropsten	51,571	626	998	49,947

In order to fully evaluate STAN, we create two kinds of bytecodes' datasets, which are shown in Table 3.5. For DS1, we crawl 42,115,551 different accounts' information in 28 days from Mainnet. We resolve all crawled accounts, with 6,920,465 accounts containing bytecodes, indicating that these are contract accounts and they are not self-destructed. After deleting all accounts containing duplicate bytecodes, we obtain 116,830 different runtime bytecodes. For DS2, we crawl open-source contracts' bytecodes from Mainnet and three public Testnets. Then we delete accounts that contain empty bytecodes, which means that they are already self-destructed, and accounts containing duplicate bytecodes. **All the bytecodes in DS2 can retrieve corresponding source codes through Etherscan, which can facilitate us to evaluate the accuracy and readability of their descriptions generated from bytecodes.** The statistics also show that only less than 1% (50,017/6,920,465) contracts are open-source.

Considering that the symbolic execution consumes time and hardware resources, we run experiments through 4 cloud instances. These instances are all configured with Intel Xeon E312x 2.60GHz CPU and 8G RAM, running 64-bit Ubuntu 18.04. We randomly extract 800 runtime bytecodes from DS1 to constitute DS1', and 200 from each network (total of 800) in DS2 to constitute DS2'. **We have checked all the bytecodes in DS1' and they are not verified with source codes in Etherscan.**

Before the evaluation, we run STAN to generate descriptions for DS1' and DS2', which include a total of 1,600 contracts' bytecodes. As a first step, we run the OYENTE [134] engine on the datasets alone, in 25 hours, to remove those contracts that encounter timeout exception. There are 651 contracts' bytecodes, 357 in DS1' and 294 in DS2', executed without timeout. Second, we generate descriptions for these 651 contracts' bytecodes through STAN, with an average analysis time of 87.4s (including symbolic execution) per contract.

3.4.2 RQ1 Adequacy

Table 3.6: Quantity statistics of the success rate of normally describing or tagging bytecodes.

DS	Bytecodes	Described	NF tagged	JE tagged	Success rate
1'	357	292 (81.8%)	62 (17.4%)	3 (0.8%)	100%
2'	294	294 (100%)	0	0	100%

In this section, we evaluate how many bytecodes can be successfully described through STAN. The quantity statistics of the success rate of normally describing or tagging bytecodes are shown in Table 3.6. Note that we tag two kinds of insecure contracts, i.e., NF (No Function) contracts, and JE (Jump Exception) contracts.

For DS1', there are 292 (81.8%) bytecodes described normally through our NLG module. The other 65 bytecodes are tagged as insecure contracts, i.e., NF (No Function) contracts or JE (Jump Exception) contracts, which are advised not to be called. The NF contract has no external/public function and executes the same opcode snippet for each invocation. For example, one tagged NF contract (Address at Mainnet: 0x5170E3C93df0605F3b02b00d8C3D9a7235fcD1Ef) is a honeypot contract, and it executes the same useless operations for each invocation. It wastes users' gas and can maliciously receive users' ETH attached in the transaction. Therefore, we tag this contract's bytecodes as *"ALERT: This is an insecure NF contract!"* The JE contract has invalid jump destination(s) in its opcodes, which may encounter jump exception and exhaust users' gas. For one tagged JE contract (Address at Mainnet: 0x6a5dffaAdBCbeF3359a017cc5100908630364aBF), regardless of which interface is called, it will encounter a runtime exception, which exhausts users' gas. Therefore, we tag this contract's bytecodes as *"ALERT: This is an insecure JE contract!"*

For DS2', all the 294 (100%) contracts' bytecodes are normally described and no bytecodes tagged as insecure contracts. We can conclude that contracts with corresponding verified source codes are generally more secure.

Table 3.7: Quantity statistics of interfaces described normally, and the success rate of four kinds of descriptions.

DS	**Interfaces**	**FD**	**UD**	**BD & PD**
1'	3,179 (N/A)	2,231 (70.2%)	2,979 (93.7%)	3,179 (100%)
2'	4,180 (100%)	3,023 (72.3%)	4,180 (100%)	4,180 (100%)

We further evaluate STAN's adequacy from the level of interfaces, whose statistics are shown in Table 3.7. For the described 292 contracts' bytecodes in DS1', 3,179 interfaces are analyzed. There are 2,231 (70.2%) interfaces' functionalities successfully described. Some interfaces are failed to generate functionality description because they are not included in DevDoc and ERCDoc analysis. In the meanwhile, they cannot be analyzed perfectly through SWUM, which are mainly reflected in two aspects. First, some functions are highly irregularly named. For example, function `caps(address)` cannot be recognized through Stanford parser [145] because "caps" is not a complete word or standard abbreviation. Second, some functions' syntax structure cannot be analyzed. For example, function `MAX_INVESTMENTS_BEFORE_CHANGE()` cannot be classified into the four syntax trees we detect. There are 2,979 (93.7%) text signatures recognized through our usage analysis and contract database. To the best of our knowledge, we already construct the most comprehensive function signature dataset. For the other 6.3% functions, there is currently no viable way to identify their text signatures.

For the 294 contracts' bytecodes in DS2', by using SCANS, we acquire their corresponding source codes and totally extract 4,180 external/public functions statically.

As shown in Table 3.7, 100% of these interfaces are analyzed and identified text signatures through usage analysis. Similar to DS1', 72.3% interfaces' functionalities successfully described.

Answer to RQ1 (Adequacy): STAN can successfully describe or tag 100% of the bytecodes in two datasets. Furthermore, 100% of interfaces can be successfully described by two description modules (i.e., BD, PD). More than 93.7% of interfaces' usage descriptions can be successfully generated, and more than 70.2% of interfaces' functionalities can be successfully described. STAN can adequately describe bytecodes of smart contracts.

3.4.3 RQ2 Accuracy

Table 3.8: Quantity statistics of STAN's accuracy of tagging insecure bytecodes.

DS	**Result**	**Accuracy**	**DS**	**Result**	**Accuracy**
1'	NF tag	62✓ / 0✗	**2'**	UD	4,180✓ / 0✗
1'	JE tag	3✓ / 0✗	**2'**	PD non-payable	2,546✓/0✗
2'	FD	217✓ / 12✢ / 1✗	**2'**	PD payable	1,634✓/0✗

In this section, we evaluate to what extent can STAN accurately describe bytecodes. We first evaluate the insecure contracts' bytecodes tagged in DS1', and FD(functionality description)/UD(usage description)/PD(payment description) in DS2', whose statistics are shown in Table 3.8. Unlike DS1', all bytecodes in DS2' have corresponding source codes, which makes it possible for us to evaluate the accuracy of their FD/UD/PD/BD(behavior description) through sources' review and static analysis.

By using DISASM [27], which is a disassembler tool, we acquire all 62 tagged NF bytecodes' corresponding opcodes and retrieve operation `PUSH4` in them. No matter which type of runtime function dispatcher, there is bound to exist `PUSH4` operation. However, `PUSH4` is not retrieved in the 62 tagged NF contracts. To further validate our conclusion, we analyze all history transactions of these 62 NF contracts. Only 2 of these 62 contracts were invoked after creation. All the 10 history transactions of one contract (Address at Mainnet: 0x84161a5491D9A9348ED48d44b2c717C9ab9 2B4F3) were reverted, which wastes users' gas, and the other contract (Address at Mainnet: 0xbB38048902107b62A680db6bA69d6d356D6A8014) maliciously received user's ETH attached in transaction. For the 3 tagged JE contracts' bytecodes, only 1 contract (Address at Mainnet: 0x9C88d1967fE2653da893B742aDa960D6570592b7) was invoked after creation, which encountered error *"Bad jump destination"*. For the other 2 contracts, we re-deploy them in our private local chain and invoke them, the same error was encountered as a result.

For the FD, we randomly select 20 bytecodes from DS2' and totally get 230 interfaces with their FDs generated by STAN. To avoid the threat of inter-rater reliability, we ask three different people to evaluate their accuracy. Through manual sources' review, we discover that 217 (94.3%) interfaces' FDs are accurate, while 12 (5.2%) interfaces' FDs are inaccurate and 1 (0.4%) interface's FD is wrong. Inaccurate and wrong FDs are mainly due to that there are incomprehensible abbreviations in some FDs. For example, FD of function `getBlockNM()` is *"Gets block nm"*.

To evaluate the accuracy of UD, leveraging SCANS, we acquire all of the 4,180 functions' text signatures from their source codes. Then we use Keccak-256 hash algorithm to calculate bytes signature for each of them. Verified by comparison with bytecodes' descriptions, 100% of the 4,180 functions' text signatures in UD are correct.

For the PD, we first acquire ABIs of all the bytecodes' corresponding Solidity sources in DS2' from Etherscan. Leveraging SCANS, we statically analyze the `payable` field (`True` or `False`) of every external/public function in ABIs. Then we compare the results to the payment descriptions generated through STAN. As a result, 100% of 2,546 non-payable and 1,634 payable interfaces, which are all described from runtime bytecodes through STAN, are correct.

We further evaluate STAN's accuracy of BD. STAN totally detects and describes 72 different interfaces with message-call behaviors, whose statistics are shown in Table 3.9 (⋆ marks pre-compiled contract calls). Leveraging SCANS, we statically analyze the AST of those functions' corresponding Solidity sources, trying to detect Solidity statements corresponding to these specific message-call behaviors. For the other described interfaces without message-call behavior, there is no related statement detected. Note that the user-defined contract call behavior has no fixed Solidity statement, and we check the 26 cases manually through source review.

Table 3.9: Quantity statistics of STAN's accuracy of message-call behaviors' description in BD.

DS	**Result**	**Accuracy**	**Evaluated statement**
2'	ETH transfer	20 ✓	transfer()/call.value()/selfdestruct()
2'	⋆ECDSA sig recovery	16 ✓	ecrecover(bytes32,uint8,bytes32,...)
2'	U-defined contract call	26 ✓	N/A
2'	⋆SHA-256 hash	2 ✓	sha256(bytes)
2'	⋆RIPEMD-256 hash	1 ✓	ripemd160(bytes)
2'	Contract deployment	7 ✓	new CONTRACT

Answer to RQ2 (Accuracy): 100% of the insecure contracts are correctly tagged, and more than 94.3% of generated functionality descriptions are correct. 100% of generated usage/payment/behavior descriptions are correct. STAN can accurately describe bytecodes of smart contracts.

3.4.4 RQ3 Readability

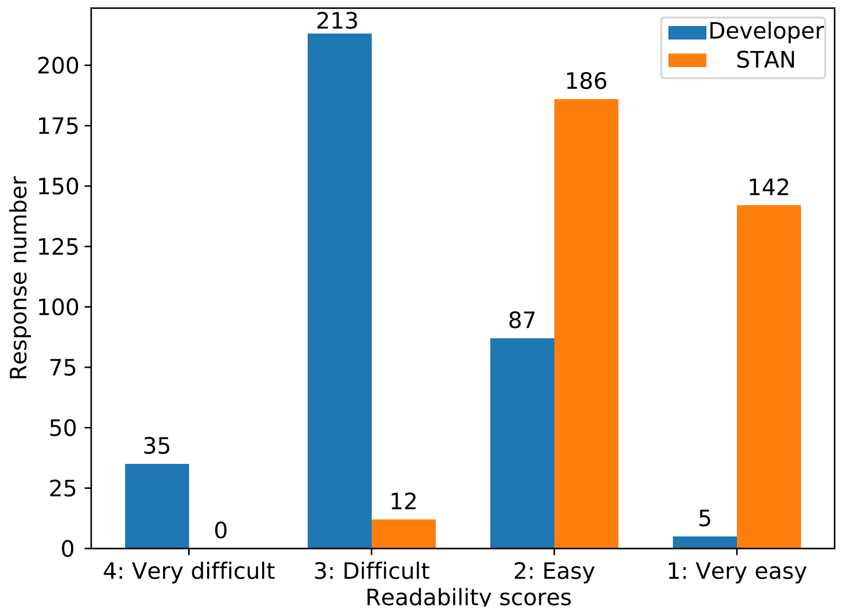


Figure 3.6: Quantity statistics of readability scores evaluated manually and their corresponding response number.

In this section, we evaluate the readability of descriptions generated through STAN. First, from the 4,180 described interfaces in DS2', we *randomly* select 10 interfaces and make sure that they all have annotations written by developers in their corresponding Solidity sources. Then we try to evaluate the readability of these 10 interfaces' descriptions generated through STAN from their bytecodes, and their annotations written by developers. Second, we design a questionnaire through SURVEYMONKEY [52]. We set a screening question only to accept those who have ever used Ethereum before. Furthermore, we set four readability scores and options (*4:very difficult, 3:difficult, 2:easy, 1:very easy*) in 20 evaluation questions. Also, if the responder thinks the annotations or descriptions difficult to read, we set open questions to input their reasons.

Third, we publish the questionnaire in BlockFlow [8], which is a blockchain development forum. During 4 days, we totally receive 38 responses. However, 2 responses are incomplete, and 2 responses do not meet the screening criteria. Therefore, the completion rate of the questionnaire is 95% (34/38). Quantity statistics of readability scores and their corresponding response numbers are shown in Figure 3.6. For total 340 evaluation responses (34 complete questionnaires for 10 interfaces), 72.9% (248/340) responses think the annotations written by developers are (very) difficult to read, while 96.5% (328/340) responses think the descriptions generated through STAN are (very) easy to read.

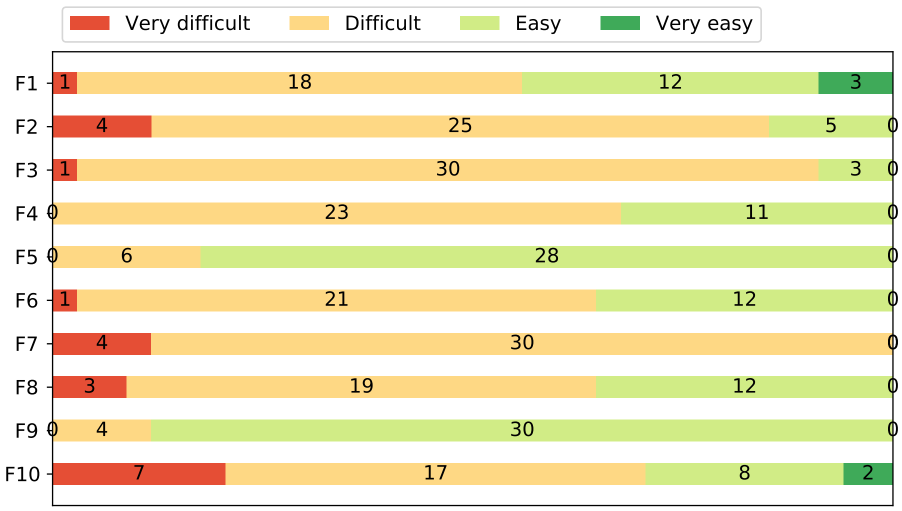


Figure 3.7: Quantity statistics of readability scores for 10 interfaces' annotations written by developers.

We also analyze the reasons that responders provide why they think some items (very) difficult to read. We further analyze and compare the readability scores

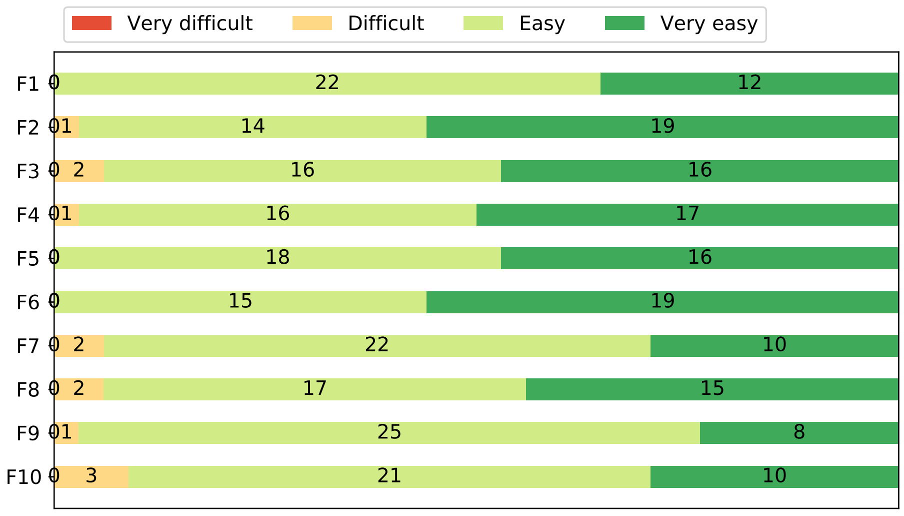


Figure 3.8: Quantity statistics of readability scores for 10 interfaces' descriptions generated through STAN.

distributions of developers' annotations and STAN's descriptions, whose quantity statistics are shown in Figure 3.7 and 3.8. For the descriptions generated through STAN, one responder suggests giving Solidity snippet example to call the bytecodes' interface, which is out of scope of this chapter. For the annotations written by developers, there are mainly 3 different reasons. There are 103 responders think the annotations are too simple explanation for interfaces, 36 responders think there exist syntax errors in annotations, and 29 responders think some vocabulary cannot be understood. Through manually checking the corresponding specific content of developers' annotations, function *0x13af4035*'s annotation only has 3 words (*"Change owner address"*), which 21 responders think is too simple. For function *0xa9059cbb*'s annotation (*"Check if the sender has enough. Add the same to the recipient."*), 16 responders think it has syntax errors. For function *0xa9059cbb*'s annotation (*"SafeMath.sub will throw"*), 12 responders think some vocabulary cannot be understood.

Leveraging Scipy [75], we further analyze the statistical distributions of the readability scores through two kinds of non-parametric tests, whose statistics are shown in Table 3.10. First, we compare the AVG (average) and STD (standard deviation) values of the readability scores for the bytecodes' descriptions generated through Stan and annotations written by developers. As a result, all of Stan's descriptions perform better than their corresponding developers' annotations in AVG. Even compared to the best AVG of developer's annotation (i.e., 2.118), Stan's descriptions are all received better scores. For STD, all of the most significant three values are appeared in developers' annotations. That is to say, the annotations written by different developers, as well as different responders for the same annotations, there exist significant differences.

Table 3.10: Statistics of non-parametric tests for the readability of STAN's descriptions.

Interface ID	**Function Bytes signature**	**AVG value Developer / STAN**	**STD value Developer / STAN**	**p value Kolmogorov-Smirnov Z**	**h value KSZ**	**p value Mann-Whitney U**	**h value MWU**
1	0xe724529c	2.500 / 1.647	**0.697** / 0.477	3.118425 * e-05	1	7.142322 * e-07	1
2	0x42966c68	2.971 / 1.471	0.514 / 0.554	7.693612 * e-12	1	5.764868 * e-12	1
3	0xa9059cbb	2.941 / 1.588	0.338 / 0.589	7.327250 * e-13	1	3.956494 * e-12	1
4	0xa9059cbb	2.676 / 1.529	0.468 / 0.543	5.118627 * e-07	1	4.887877 * e-10	1
5	0x4bb278f3	2.176 / 1.531	0.381 / 0.499	8.952478 * e-04	1	6.089033 * e-07	1
6	0x18160ddd	2.676 / 1.441	0.527 / 0.497	5.118627 * e-07	1	1.067191 * e-10	1
7	0xd73dd623	3.118 / 1.764	0.322 / 0.546	1.601234 * e-16	1	2.134156 * e-13	1
8	0xb602a917	2.735 / 1.617	**0.609** / 0.594	8.662292 * e-06	1	8.142437 * e-09	1
9	0x74a8f103	**2.118** / 1.794	0.322 / 0.471	**3.067583 * e-01**	**0**	1.129570 * e-03	1
10	0x13af4035	2.853 / 1.792	**0.809** / 0.583	2.204947 * e-06	1	2.625890 * e-07	1

Second, we conduct Kolmogorov-Smirnov Z and Mann-Whitney U tests to detect whether the two sets of scores have the same statistical distribution. If the `p` value is less than 0.05, which is a relatively strict threshold, the result hypotheses value will be 1, and the two sets' statistical distributions are different. As a result, except for interface ID-9's KSZ test, all results show that the two sets of scores have different statistical distributions. Through manual checking, we discover that the annotations of ID-9 perform the best in the 10 samples, which receive scores closest to those of STAN's descriptions.

Answer to RQ3 (Readability): Compared with the interfaces' annotations written by developers, 96.5% manual responses think the descriptions generated through STAN are (very) easy to read. Furthermore, STAN can generate more stable and readable descriptions than developers' annotations.

3.5 Limitations and Solutions

In this section, we discuss some limitations and the corresponding solutions, which are as follows:

(1). As described in Section 3.4.1, we run the symbolic execution engine alone on two datasets to discover that many contracts' bytecodes encounter timeout exception. In future work, we will improve cloud instance's configuration, and use more significant timeout threshold to reduce the number of timeout cases.

(2). We analyze every execution path and some opcodes' symbolic values to describe interfaces accurately and comprehensively, with an average analysis time of 87.4s per contract. In future work, we may consider improving STAN's performance with faster static analysis techniques and evaluating STAN with more comprehensive bytecodes datasets.

(3). As described in Section 3.4.2, some functions' signatures cannot be analyzed perfectly through SWUM. In future work, we will build more and better syntax trees, and add more common word abbreviations in Ethereum to Stanford parser's rule libraries to improve SWUM analysis.

(4). Stan can generate four categories of descriptions for each interface, as well as tag two kinds of insecure contracts' bytecodes. In future work, we will conduct more features' and behavior' analysis to improve Stan's functionalities.

3.6 Brief Summary

In this chapter, we propose Stan, which leverages symbolic execution and NLP techniques to describe runtime bytecodes of smart contracts. Stan can generate four categories of descriptions in natural language for every interface of bytecodes deployed in Ethereum. We also develop static tool Scans to facilitate us to construct the database for Stan, and facilitate us to evaluate the generated descriptions. Extensive experiments show that Stan can generate adequate, accurate, and readable descriptions for bytecodes. In future work, we will explore other techniques (e.g., machine learning [177] [171]) to generate better descriptions.

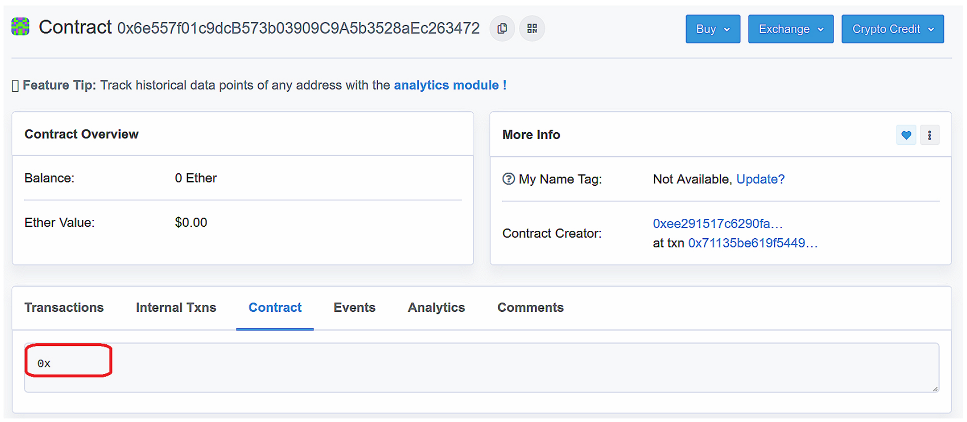
Chapter 4

Analysis of Erasable Accounts in Ethereum

Being the most popular permissionless blockchain that supports smart contracts, Ethereum allows any user to create accounts on it. However, not all accounts matter. For example, the accounts due to attacks can be removed. In this chapter, we conduct the first investigation on erasable accounts that can be removed to save system resources and even users' money (i.e., ETH or gas). In particular, we propose and develop a novel tool named GLASER, which analyzes the State DataBase of Ethereum to discover five kinds of erasable accounts. The experimental results show that GLASER can accurately reveal 508,482 erasable accounts and these accounts lead to users wasting more than 106 million dollars. GLASER can help stop further economic loss caused by these detected accounts. Moreover, GLASER characterizes the attacks/behaviors related to detected erasable accounts through graph analysis.

4.1 Overview

Being the largest blockchain that supports smart contract, Ethereum has two kinds of accounts: EOA (Externally Owned Account) and contract account [132]. As a permissionless blockchain system, Ethereum allows any user to create many EOAs through their private keys. Deploying a smart contract to Ethereum will produce a contract account that contains the contract's runtime bytecodes. Every node must synchronize blockchain data, which includes blocks and StateDB (State DataBase) [62]. The StateDB stores all the accounts' state information, such as ETH balance, transaction number, runtime bytecodes, and so on [62].



Contract 0x6e557f01c9dcB573b03909C9A5b3528aEc263472

Feature Tip: Track historical data points of any address with the **analytics module !**

Contract Overview	
Balance:	0 Ether
Ether Value:	$0.00

More Info	
My Name Tag:	Not Available, Update?
Contract Creator:	0xee291517c6290fa... at txn 0x71135be619f5449...

```
0x
```

Figure 4.1: One empty account detected by GLASER.

However, not all accounts should be kept. In particular, we identify three kinds of erasable contract accounts that are produced due to contracts' programming errors or attacks, and two kinds of erasable EOAs that are produced due to contracts' deployment failure or DoS (Denial of Service) attacks. Such erasable accounts not only waste system resources and affect the efficiency of blockchain, but also easily waste users' money (i.e., ETH or gas). For example, one empty account (Address: 0x 6e557f01c9dcb573b03909c9a5b3528aec263472) discovered in this chapter was created due to contract deployment failure. It wasted user's 137,552 gas when it was called because the contract's runtime bytecodes were not stored in this account, whose information is shown in Figure 4.1. We regard the worthless accounts that deserve to be removed without affecting the normal operations of users and other accounts as erasable accounts.

Unfortunately, there lacks a systematic study on the erasable accounts that can be removed. Although some studies [85] [84] use call graph analysis to measure the control flow between contracts, their purposes are different from ours. Our work focuses on the erasable accounts that exist in Ethereum, and some of our analyzed accounts (e.g., DoS contracts) are related to interaction between contracts. There also exist some other research analyzing different kinds of security issues for smart contracts [102] [86] or Ethereum architecture [63] [173] [135]. These research mainly focus on security issues on the contract-level and system-level of Ethereum, whose contents and purposes are different from ours.

To fill the gap, we design and implement a novel tool named GLASER (detectinG erasabLe AccountS in EtheReum) to discover erasable accounts by analyzing the StateDB of Ethereum. It is worth noting that marking an account as erasable just according to its liveness and balance value is improper, because an account might contain useful runtime bytecodes or its private key is owned by external user so that it cannot be removed even if it has not been used for a long time and stores no ETH. Instead, GLASER analyzes accounts' contents and states stored in Ethereum StateDB. In detail, it leverages program analysis techniques to discover contract accounts with worthless runtime bytecodes, and employs state field and transaction analysis to discover EOAs that no one owns their private keys. The accounts discovered by GLASER are worthless and deserve to be removed without affecting the normal operations of other accounts/users.

Applying GLASER to all Ethereum accounts, we discovered 508,482 erasable accounts, and more than 99.9% of them are still stored in Ethereum. These erasable accounts have wasted users more than 106 million dollars and can be removed through executing `SELFDESTRUCT` operation in their runtime bytecodes by users, or removed forcibly by Ethereum officials. For example, one erasable contract account (Address: 0xa30BCeA7E5806aC5D37D221D2F8A40642B0Bb1a6) can be removed through transaction sent by any user, and some empty account created due to DoS attacks were already removed forcibly through hard fork by Ethereum officials [15]. This chapter mainly focuses on erasable accounts' detection to help users identify erasable accounts and remind users not to call them to save money, and erasable accounts' characterization to interpret their behaviors/attacks and creation reasons.

Our Contributions. The main contributions of this chapter are listed as follows:

- To the best of our knowledge, we conduct the *first* systematic investigation on erasable accounts in Ethereum. We propose and define five kinds of erasable accounts, i.e., three kinds of erasable contracts and two kinds of erasable EOAs.

- We design a novel approach to analyze the Ethereum StateDB, and implement the idea in a tool called GLASER, which can discover and characterize erasable contract accounts and erasable EOAs. For contract accounts, leveraging static analysis and symbolic execution, GLASER analyzes runtime bytecodes of contracts to detect three kinds of erasable contract accounts. For EOAs, GLASER analyzes their state-related attribute fields and historical transactions to discover two kinds of erasable EOAs. GLASER also characterizes erasable accounts through call graph and creation graph analysis.

- We conduct experiments to evaluate and characterize the detected erasable accounts. We analyze the 508,482 detected erasable accounts' creation time distributions. More than 99.9% of them are still stored in Ethereum, and their transactions wasted users more than 106 million dollars. GLASER can remind users not to call erasable accounts and help stop further economic loss of users caused by them. Furthermore, the graph analysis of erasable accounts interprets their creation reasons, i.e., attacks, programming errors, or deployment failure.

This chapter is organised as follows. Section 4.2 introduces the technical background and Section 4.3 interprets the details of erasable accounts. Then Section 4.4 details the principles of GLASER. Section 4.5 evaluates the results and Section 4.6 characterizes attacks/behaviors related to discovered erasable accounts. At last, we discuss this chapter in Section 4.7 and conduct summarization in Section 4.8.

4.2 Background

We briefly introduce the knowledge involved in this chapter.

4.2.1 StateDB

Supporting smart contracts, Ethereum records not only transactions but also state transitions that occur in blockchain. Ethereum contains two types of accounts, i.e., EOA and contract account [164], which are all indexed by 20 bytes length of addresses.

Account's creation and usage: Ethereum is a permissionless blockchain system, and users can create their own EOA and store ETH (native cryptocurrency in Ethereum). Users can initiate transactions by the private key corresponding to the EOA address, including ETH transfers and contract calls. The contract accounts are created by EOAs or other contract accounts. In addition to storing ETH, the contract account also holds the runtime bytecodes of smart contract. There are two types of bytecodes in Ethereum: runtime bytecodes stored in contract account, and deployment bytecodes used for contract runtime bytecodes' deployment. The contract account is not controlled by the user's private key, but by the contract's runtime bytecodes' logics.

Account's removal: Users can only remove contract account through executing `SELFDESTRUCT` in its runtime bytecodes. All EOAs and contract accounts without `SELFDESTRUCT` in runtime bytecodes cannot be removed by users. In addition, all erasable accounts can be removed forcibly by Ethereum officials. Although some discovered erasable accounts in this chapter cannot be removed by users, our results can remind users not to call them to save money.

StateDB: The StateDB stores the world state of Ethereum based on accounts. For every account a, its state $\sigma[a]$ consists of four fields [62]: If a is an EOA, $\sigma[a]_n$ stores the number of external transactions *sent from* this account. If a is a contract account, $\sigma[a]_n$ stores the number of contracts created by this account. $\sigma[a]_b$ stores the balance value (in Wei) of account a. $\sigma[a]_s$ stores the root hash of Merkle tree which encodes the storage contents of the account. $\sigma[a]_c$ stores the runtime bytecodes of account a. Note that the main difference between EOA and contract account is whether its code field is empty [62].

4.2.2 Contract Execution

When a smart contract is deployed in Ethereum, users can invoke its external functions through transactions. Note that we describe transactions sent from EOAs as external transactions, and message-calls sent from contract accounts as internal transactions in this chapter. Gas is the basic unit of resource consumption for transactions in Ethereum [178]. Before users initiate transactions, they all need to pay a certain amount of gas. When the smart contract is running in EVM, each opcode corresponds to a certain amount of gas, whose value is defined in the Ethereum Yellow Paper [62]. To prevent DoS attacks, Ethereum has modified the gas value of some specific opcodes, such as `SELFDESTRUCT`'s value was modified from 0 to 5,000 in EIP-150 (Ethereum Improvement Proposal) [79].

The smart contracts' execution in EVM involves three forms of data, namely storage, memory, and stack [80]. The storage data is stored in StateDB of Ethereum in the form of key-value pairs, and both key's length and value's length are 256 bits [132]. Storage is persistent and will not be released as transaction execution ends. Storage data is stored and read through two opcodes, i.e., `SLOAD` and `SSTORE`. Memory is the temporarily allocated space when smart contracts are executed in EVM, which is automatically freed as the transaction execution finishes. EVM is a 1,024 depth stack-based virtual machine, and the contracts' opcodes are all executed around the stack [131].

4.2.3 Account Analysis

Frowis et al. [103] constructed call graph for smart contracts deployed in Ethereum and discovered contracts calling to removed contracts. Note that they focus on measuring the control flow immutability between contracts, whose purpose is different from our work. Kiffer et al. [126] measured smart contracts' creation and interaction with each other, which interpreted how are smart contracts being used. However, they do not analyze erasable contracts that exist in Ethereum. Kiffer et al. [126] measured the overall usage of Ethereum, which interpreted how is Ethereum being used. They discovered that `SELFDESTRUCT`'s usage rose sharply during DoS attacks in 2016. However, they do not measure or analyze erasable accounts produced during DoS attacks. Chen et al. [81] proposed an adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks. They do not analyze real accounts in Ethereum that are related to these attacks. Wang et al. [167] proposed an optimization storage engine to reduce nodes' storage volume, which can improve the scalability of blockchain systems. They do not analyze the erasable accounts which are already stored in StateDB. Angelo et al. [94] analyzed contract deployment code patterns which were exploited by attackers, and they described three related attack scenarios in reality appeared in the middle of 2018, whose contents and purposes are different from ours. They focus on the vulnerabilities and attacks leveraging skillfully crafted deployment codes, while we detect erasable accounts due to programming or deployment errors.

4.3 Erasable Accounts

We introduce erasable accounts detected in this chapter.

4.3.1 Erasable Contract

The main difference between EOA and contract account is whether its code field is empty [62]. Below we introduce erasable contracts with runtime bytecodes.

Meaningless contract: We analyze two kinds of meaningless contract, i.e., MC-S (Meaningless Contract with `STOP`) and MC-RS (Meaningless Contract with `REVERT` or `SELFDESTRUCT`).

MC-S refers to one particular kind of meaningless contract, whose first opcode in its runtime bytecodes is `STOP`. There exist MC-S because users incorrectly use runtime bytecodes to deploy contracts, whose creation and behavior will be analyzed in Section 4.5. When the MC-S is called, `STOP` will halt the transaction's execution immediately. Therefore, these contracts are controlled by `STOP`, which is meaningless and may waste user's gas or ETH.

MC-S Example: One MC-S (Address: 0x2Ab748a546760b1EC834E164DEDE2E7 1C4010E1d) was called with input data three times, which waste users' gas. Their input data were not processed before the related transactions were halted by `STOP`. Furthermore, this meaningless contract was transferred ETH through transactions twice. Because the MC-S is controlled by `STOP`, the total of more than 0.042 ETH stored in this account can never be transferred out, which results in users' money waste.

MC-RS refers to contract that has `REVERT` or `SELFDESTRUCT` opcode in its first basic block. A basic block means a series of sequential opcodes without any control flow operation (e.g., `JUMP`, `STOP`) [79]. The first basic block is the program entrance and every call to the contract will execute it. Most MC-RS are deployed by malicious contracts through internal transactions (i.e., sent from contract). However, MC-RS is meaningless because any call to MC-RS will invoke `REVERT` or `SELFDESTRUCT`. `REVERT` ends runtime bytecodes' execution and reverts state changes of the call. `SELFDESTRUCT` removes the contract account from blockchain.

```
0x0  PUSH1 0x00  //MC-RS-1
0x2  CALLDATALOAD  //get the first 32bytes call data
0x3  SELFDESTRUCT  //destruct the contract
-------------------------------------------------------
0x0  PUSH1 0x80  //MC-RS-2
0x2  PUSH1 0x40
0x4  MSTORE  //save 0x80 to memory
0x5  PUSH1 0x00
0x7  DUP1
0x8  REVERT //end execution and revert state changes
0x9  STOP
0xa  LOG2  //other basic blocks
…… ……
```

Figure 4.2: Snippets of two MC-RS.

MC-RS Example: The snippets of two MC-RS are shown in Figure 4.2. There are only three operations in the first MC-RS (Address: 0xa30BCeA7E5806aC5D3 7D221D2F8A40642B0Bb1a6). This contract can be exploited by attacker to steal ETH through setting his own EOA address in the call data. However, this contract is meaningless. Because any call to it will invoke `SELFDESTRUCT` and transfer out the ETH stored in it. The second MC-RS (Address: 0x7770A80851A266e717dC93 A194A7eC0875214293) will invoke `REVERT` during any call to it. Furthermore, the operations after its first basic block will never execute. Any call to the contract will execute operations from `0x0` to `0x8`, which is meaningless and waste gas.

Stack or opcode error contract: EVM is a virtual machine based on 1,024 depth stack, and the stack will definitely overflow (push more than 1,024 items into stack) or underflow (pop item from empty stack) if the stack error contract is called. Before EIP-150 increased the gas cost of `CALL` from 40 to 700, the attacker may exploit stack overflow through recursive call depth attack [132]. Nowadays, although stack overflow is hard to occur, stack underflow still exists due to program writing errors.

Stack Error Contract Example: One transaction (Hash: 0x9518bcde68b522a4521c3eeade8fa461af16b5c7f0d1529d7ead27663d4e5092) encountered "Stack Underflow" error and exhausted its gas, due to its contract deployment-related codes. Moreover, runtime bytecodes' contents may be related to some uncontrollable factors, which may also produce stack error contracts. One example of stack error contract's deployment bytecodes is shown in Figure 4.3 (Related transaction: 0xf7db99fb4524133839915b8e08914dae0f9bcecd6847691e4dd2ce8ead61e420). In program counter `0x5`, it returns runtime bytecodes to deploy, whose first byte is related to the current block's timestamp (in program counter 0x0 to 0x1). At last, one stack error contract (Address: 0x7a0352aa3231d2255a96113b619057994341069e) was deployed, and its first operation in runtime bytecodes is `DIV`, which will result in stack underflow.

Developers can use high-level languages or directly write bytecodes to develop smart contract. However, due to programming errors, some runtime bytecodes deployed in blockchain cannot be disassembled to correct opcodes. If there exist unknown opcodes that cannot be recognized by EVM, it will encounter "Bad Instruction Error". Opcode error contract refers to contract that has unknown opcode in the first

```
0x0 TIMESTAMP //get the block's timestamp
0x1 CALLVALUE
0x2 MSTORE8 //save byte to memory
0x3 CODESIZE
0x4 CALLVALUE
0x5 RETURN //return bytecodes to deploy
```

Figure 4.3: The deployment bytecodes of one stack error contract.

basic block, which will encounter error during any call to it.

<u>Opcode Error Contract Example:</u> The first two bytes of one contract's runtime bytecodes are `0xd929`, which cannot be correctly disassembled to opcodes of EVM. Because the unknown opcodes exist in its first basic block, all transactions calling to it encountered "Bad Instruction Error" (Address: 0x526634cde83e541ba851a402e5c85bd 0838505eb) (need to be viewed in advanced mode in Etherscan). The transaction with "Bad Instruction Error" exhausts gas, halts execution and reverts state changes [62].

DoS contract: We analyze two kinds of DoS related contracts: attacked Parity wallets, and malicious contracts exploited for DoS attacks. If contract `A` hardcodes and calls contract `B`'s address to execute, and `B` is removed, `A` will be a dependency error contract without normal service. In November of 2017, the attacker escalated his privilege and removed Parity's multi-sig library contract (Address: 0x863df6bfa4 469f3ead0be8f9f2aae51c91a907b4), which caused all Parity wallets that depend on it out of service. Note that calling to a removed contract will just return 1 (means no error or exception), and users cannot verify if it is out of service through return value. If users knew in advance that their wallets were out of service, they would not use them anymore to deduce financial losses. Etherscan only marks part of attacked Parity wallets, we attempt to detect more of them.

In 2016, the attacker exploited malicious contracts to initiate DoS attack for Ethereum [132]. The attacker executes massive particular operations (e.g., `EXTCODESIZE`, `DELEGATECALL`), which consume low gas but high system resources. The DoS attack leads to low nodes' data synchronization and transaction execution. The Ethereum official modified many operations' gas values in EIP-150 [132] to repair related vulnerabilities.

Malicious DoS Contract Example: We analyze the malicious DoS contracts and discover that they have similar patterns. These malicious contracts just have one basic block in their runtime bytecodes. In the basic block, there are many particular operations that consume low gas but high system resources. For example, one malicious DoS contract's snippets are shown in Figure 4.4 (Address: 0x792218d8bbe00fb81296236b014Fb14af2DA385B) with 200 `EXTCODESIZE` in the only basic block of the contract.

```
0x00 PUSH20 0x42a119d24fd64362f3892815d310c83edcb61b88
0x15 EXTCODESIZE
0x16 POP
0x17 PUSH20 0xdfccc8e473dc262cfc6ddb4092946b66baadf88b
0x2c EXTCODESIZE //its gas was modified from 20 to 700
0x2d POP
0x2e PUSH20 0xd96b74abd2ded0b7f2873202a2f3bb562b22b2ef
0x43 EXTCODESIZE
0x44 POP
```

Figure 4.4: Snippets of one malicious DoS contract.

4.3.2 Erasable EOA

Below we detect erasable EOAs, which do not store codes.

Empty account: The empty account has the following features: ❶zero value balance, ❷zero value nonce, and ❸empty code. Whether the account has code is the main difference between contract account and EOA, and we classify empty accounts into erasable EOA. Ethereum officials have only cleaned up the empty accounts created during the DoS attack exploiting `SELFDESTRUCT` [15]. However, there still exist empty accounts due to contract deployment failure. Before EIP-2, it will create an empty account if the contract deployment transaction does not succeed (e.g., out of gas). After EIP-2, it will fail with error and do not create empty accounts anymore. The creation process of empty accounts denotes that they are not controlled by runtime bytecodes or external users, which results in their uselessness. The empty account may result in gas waste, because users may incorrectly think runtime bytecodes are deployed in these accounts.

Example: One empty account (Address: 0x6e557f01c9dcB573b03909C9A5b352 8aEc263472) has been called many times, which wastes users' gas. We analyze all the input data of the related transactions, whose first four bytes are all function signatures. That is to say, all these transactions were intended to invoke the functions in runtime bytecodes.

DoS EOA: The DoS EOA has the following features: ❶1 Wei value balance, ❷zero value nonce, ❸empty code, ❹zero historical external transaction, and ❺one historical internal transaction without error. The differences between empty account and DoS EOA are their balance value and creation process. DoS EOAs are created through internal transactions sent from contracts. Massive DoS EOAs were created during the DoS attack in 2016, whose creation will be analyzed in Section 4.5.1. The attacker created DoS EOAs through smallest financial cost (i.e., 1 Wei), and all of these accounts' addresses were generated through computation in runtime bytecodes, whose process denotes their uselessness (detail in Section 4.6). The existence of massive DoS EOAs increases the StateDB size, resulting in the waste of disk resources and nodes' difficulty in syncing data.

Example: One transaction (Hash: 0x1aa87a25df792f1dacacbc194e3963a0cbc f950ede1d60e679500b40d9589b17) detected by GLASER created ten DoS EOAs through internal transactions. Note that 1 Wei (1 ETH = 10^{18} Wei) is the smallest cryptocurrency unit in Ethereum, which cannot even buy 1 gas. The recommended gas price is 61 GWei [24] (1 GWei = 10^9 Wei), which can be set in transaction by users.

4.4 GLASER's Implementation

To analyze the StateDB, we synchronize the blockchain with "fat-db=on" option through Parity client, which can build appropriate information to allow enumeration of all accounts. Then we export the StateDB as plain text file through Parity and leverage GLASER to traverse StateDB data to detect erasable accounts. The overview of GLASER's architecture is shown in Figure 4.5, which mainly consists of three modules:

(1) Erasable contract account detection. In this module, GLASER detects three kinds of erasable contract accounts: meaningless contracts, stack/opcode error contracts, and DoS contracts. According to their respective characteristics, we leverage different techniques, which mainly include runtime bytecodes' static analysis and symbolic execution.

(2) Erasable EOA detection. In this module, GLASER detects two kinds of erasable EOAs: empty accounts, which are produced due to contract deployment failure; and

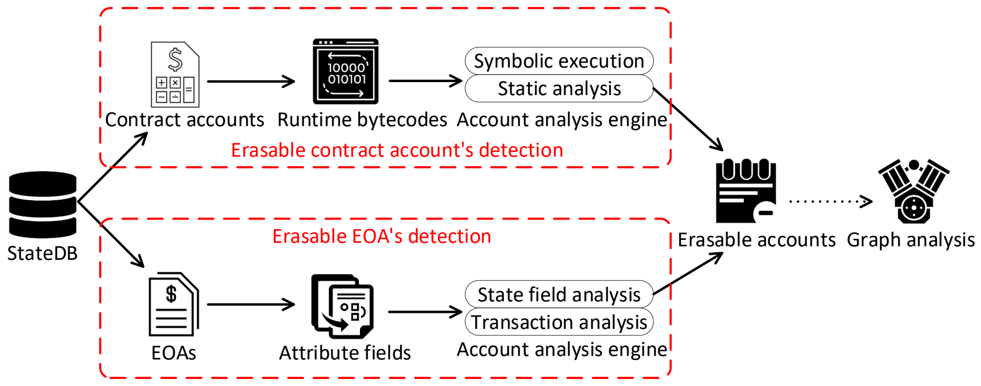


Figure 4.5: Overview of GLASER's architecture.

DoS EOAs, which are produced due to DoS attacks. We mainly leverage state field and transaction analysis to discover erasable EOAs.

(3) Graph analysis for erasable accounts. For the detected erasable accounts, GLASER characterizes their behaviors/attacks through call graph analysis and creation graph analysis, whose details will be described in Section 4.6.

4.4.1 Erasable Contract Detection

Meaningless contract: GLASER leverages runtime bytecodes' static analysis to detect two kinds of meaningless contract, i.e., MC-S and MC-RS. Static analysis refers to techniques that examine codes without attempting to execute them [88]. GLASER statically analyzes contracts' runtime bytecodes to detect MC-S. In detail, it intercepts runtime bytecodes' first byte to judge whether it is `0x00`, which is the hex code for `STOP`. If one contract starts with `0x00` byte in its runtime bytecodes, it will be tagged as MC-S. GLASER also statically analyzes contracts' runtime bytecodes to detect MC-RS. First, it disassembles contract's runtime bytecodes to acquire the opcodes. Second, it splits the opcodes into different basic blocks, which end with specific control flow operations (i.e., `STOP`, `JUMP`, `JUMPI`, `RETURN`, `SELFDESTRUCT`, `REVERT`). Third, it analyzes the first basic block. If `REVERT` or `SELFDESTRUCT` exists in the first basic block, it will tag the contract as MC-RS.

Stack or opcode error contract: GLASER leverages symbolic execution and runtime bytecodes' static analysis to detect stack/opcode error contracts. Symbolic execution uses symbolic values as inputs to simulate the process of program execution [134]. The detection process of stack error contract is divided into three steps. First, GLASER acquires the opcodes of contract's runtime bytecodes. Second, it splits the opcodes into different basic blocks and extracts the runtime bytecodes corresponding to the first basic block. Third, it symbolically executes the extracted runtime bytecodes leveraging OYENTE [134], which is a symbolic execution engine. If the symbolic execution process encounters "Stack Underflow", it will tag the contract as stack error contract. For opcode error contract, GLASER disassembles contract's runtime bytecodes into opcodes and split them into basic blocks. Then GLASER detects whether there exist unknown opcodes in its first basic block. If unknown opcode exists in its first basic block, the contract will be tagged as opcode error contract.

DoS contract: GLASER leverages symbolic execution and runtime bytecodes' static analysis to detect DoS contracts. GLASER detects attacked Parity wallet contracts leveraging symbolic execution techniques. GLASER analyzes four related operations for contracts' interaction, i.e., `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`. If the symbolic execution encounters anyone of these operations, it extracts the second item of the stack $\mu_s[1]$, which is used as the address of contract being called. If $\mu_s[1]$ is a real value that matches the address of removed Parity multi-sig library, it will tag this contract as attacked Parity wallet. For malicious DoS contract, GLASER disassembles contract's runtime bytecodes into opcodes and split them into basic blocks. Then GLASER analyzes the number and content of basic block. If one contract has only one basic block and has more than 100 DoS related operations, GLASER will tag it as malicious DoS contract. We analyze seven DoS related operations: `EXTCODESIZE`, `EXTCODECOPY`, `BALANCE`, `CALL`, `DELEGATECALL`, `CALLCODE`, `SELFDESTRUCT`.

4.4.2 Erasable EOA Detection

Empty account: GLASER leverages account state field analysis and transaction analysis to detect empty accounts. The detection process of empty accounts is divided into two steps. First, GLASER analyzes the account attribute fields to detect possible empty accounts, which should satisfy the three features described in Section 4.3.2. Second, GLASER analyzes the historical transaction of the detected empty accounts in the first step, to verify that they are created due to contract deployment failure. In detail, it analyzes the oldest transaction related to the accounts detected in the first step. If one account's oldest transaction is used for contract deployment, GLASER will tag it as erasable empty account.

DoS EOA: GLASER leverages account state field analysis and transaction analysis to detect DoS EOA. Similar to the detection of empty accounts, detection process of DoS EOAs is divided into two steps. First, GLASER analyzes the account attribute fields to detect possible DoS EOAs, which should satisfy the first three features described in Section 4.3.2. Second, GLASER analyzes the historical transaction of the detected DoS EOAs in the first step, to verify that they are created through internal transactions sent from contracts. In detail, we set relatively strict conditions to verify

DoS EOAs in this step. Glaser analyzes their historical external transactions and internal transactions. If one account has no external transaction and only one internal transaction without error (i.e., sent 1 Wei to create this account), we can conclude that it is an erasable DoS EOA. There might exist massive internal transactions with "Out of Gas Error", which were used for DoS attacks.

4.5 Evaluation

We carry out experiments to answer the following research questions:

RQ1 Quantity: How many each kind of erasable accounts can be detected through Glaser?

RQ2 Accuracy: To what extent can Glaser accurately detect erasable accounts?

RQ3 Waste: How much money lost due to erasable accounts?

4.5.1 RQ1 Quantity

In this section, we evaluate the quantity statistics of erasable accounts detected through Glaser. Furthermore, we analyze the creation time distribution of the detected erasable accounts.

Table 4.1: Quantity statistics of erasable accounts detected through Glaser.

Cat.	Taxonomy	Quantity	Erasable accounts	Quantity
❶	Erasable contract	481,087	Meaningless contract	479,153
			Stack/opcode error contract	150
			DoS contract	1,784
❷	Erasable EOA	27,395	Empty account	195
			DoS EOA	27,200

We have exported the StateDB of Ethereum and detect erasable accounts leveraging GLASER, whose quantity statistics are shown in Table 4.1. We discover 481,087 erasable contracts and 27,395 erasable EOAs respectively. All the five specific kinds of detected erasable accounts' addresses are published on `https://figshare.com/articles/dataset/11516694`. For the 1,784 DoS contracts, we detect 658 different contracts hardcode and call the removed Parity multi-sig library, while Etherscan only tags 153 of them. Because most users leverage high-level languages to develop contracts, there exists a small quantity of stack/opcode error contracts. Because Ethereum officials have already repaired the bug of empty account's creation due to contract deployment failure, the discovered empty accounts' quantity is small.

To measure the number of erasable accounts at different time, we analyze their historical transactions to acquire their creation time. The analysis of accounts' creation time is divided into two steps. First, we crawl all the historical transactions related to the detected erasable accounts through Geth RPC APIs. Second, we filter out the oldest transaction of each account and acquire the timestamp of this transaction, which is the creation time of this account.

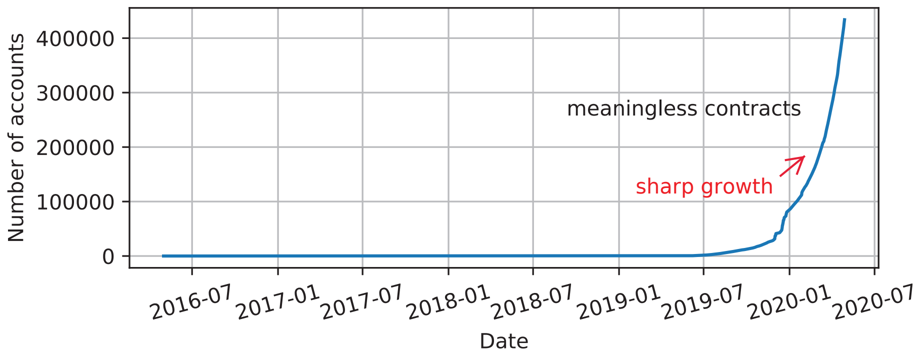


Figure 4.6: Cumulative quantity distribution of meaningless contracts.

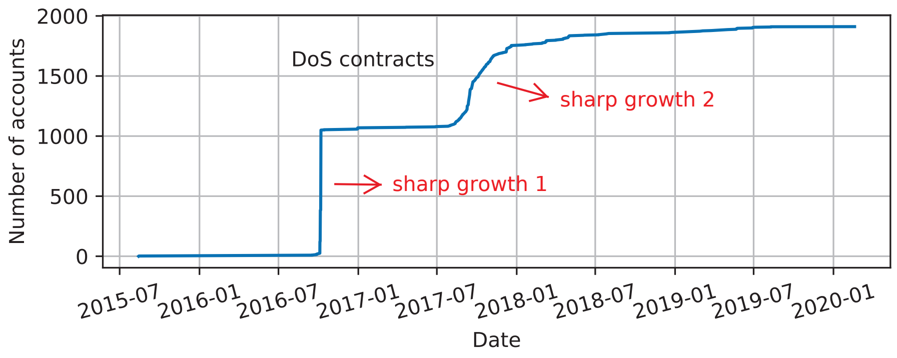


Figure 4.7: Cumulative quantity distribution of DoS contracts.

The cumulative quantity distribution of meaningless contracts at different time is shown in Figure 4.6. Before July of 2019, the quantity of meaningless contracts is small. Because most meaningless contracts are MC-S and they are directly created by users through EOA. For example, one user (Address: 0x3ff51120D34f4318B6 aff85DbCa5481DbF03f40B) created 9 MC-S with totally same runtime bytecodes around February of 2018. When the user realized his irrational behavior, he did not create MC-S any more. After November of 2019, some active malicious contracts are massively called, which leads to the quantity sharp growth of meaningless contracts (i.e., MC-RS). For example, one Ponzi contract (Address: 0x7C20218efC2e07C8Fe25 32fF860D4A5d8287cB31) created many MC-RS before April of 2020 through internal transactions (i.e., sent from contract account). Because most users leverage high-level languages to develop contracts, there exists a small quantity of stack/opcode error contracts. Their deployment time distribution does not have clear trends or characteristics.

The cumulative quantity distribution of DoS contracts at different time is shown in Figure 4.7. There are two sharp growth periods for DoS contracts. The first period is around October of 2016, the attacker deployed more than 1k malicious DoS contracts and sent massive transactions to them, leading to external transactions' slow execution. The second period is around November of 2017, the Parity's multi-sig library contract was attacked and removed during this period, which produced 658 dependency error wallets without service.

The cumulative quantity distribution of empty accounts at different time is shown in Figure 4.8. Because the Ethereum officials have repaired the bug of empty accounts' creation due to contract deployment failure and cleaned up the empty accounts produced due to DoS attacks, the growing of their cumulative quantity is halted around March of 2016. The cumulative quantity distribution of DoS EOAs is shown in Figure 4.9. There is a sharp growth period of DoS EOAs' quantity around November of 2016. According to analysis, the attacker (One exploited account: 0xeec 2a1ee6ee942596b6e255d24d38c0a9338cfef) created massive DoS EOAs during/after the DoS attacks of empty accounts' creation exploiting `SELFDESTRUCT` [15].

Answer to RQ1 (Quantity): We have discovered 508,482 erasable accounts, whose quantity distributions at different time reflect their creation reasons.

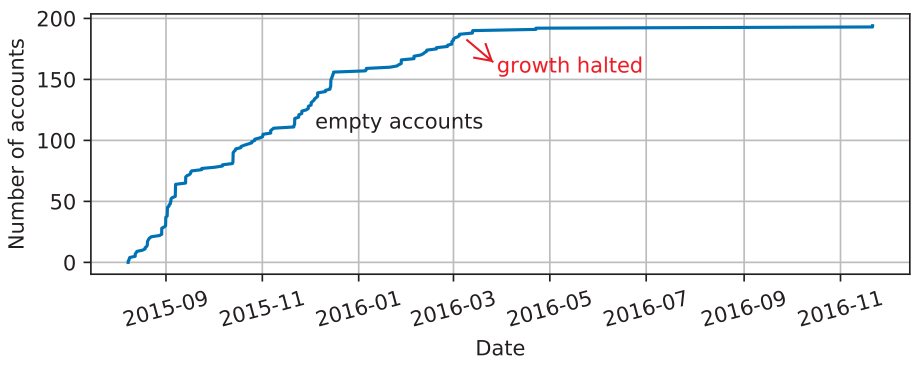


Figure 4.8: Cumulative quantity distribution of empty accounts.

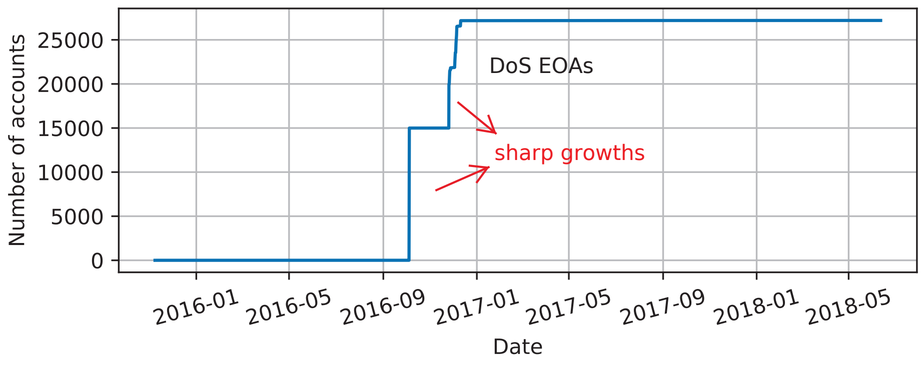


Figure 4.9: Cumulative quantity distribution of DoS EOAs.

4.5.2 RQ2 Accuracy

In this section, we evaluate the accuracy of erasable accounts detected through Glaser, whose statistics are shown in Table 4.2. We evaluate the accuracy of erasable accounts in two primary aspects. First, we analyze whether the detected erasable accounts are still stored in Ethereum. Second, we analyze their transactions to verify their uselessness.

Table 4.2: Statistics of erasable accounts' accuracy and waste evaluation.

Erasable account	**Quantity**	**Storage**	**Ext. tr.**	**Int. tr.**	**Gas**	**ETH**
❶ DoS contract	1,784	100% ✓	26,474	7,707,646	50,497,619,162	515,035.16ETH
❷ Meaningless contract	479,153	99.9% ✓	2,080	490,611	36,996,614,413	274.97ETH
❸ Stack/opcode error cont.	150	100% ✓	141	157,513	854,099,555	0
❹ Empty account	195	100% ✓	237	5	79,786,061	0
❺ DoS EOA	27,200	100% ✓	0	1,163,763	1,180,693,660	27,200 Wei

Storage: Because it is difficult to traverse accounts in its changing StateDB, we export the StateDB to offchain and execute Glaser on it. Therefore, there exists possibility that the detected erasable accounts are already removed or cleaned up in the newest StateDB. We leverage Etherscan, which is a real-time Ethereum block explorer, to verify their existence. We discover that 99.9% (479,150/479,153) detected erasable contract accounts still store runtime bytecodes. 3 MC-RS contracts are removed through executing `SELFDESTRUCT`. All the 195 empty accounts can be still normally retrieved without special tagging, and all the 27,200 detected DoS EOAs still store ETH. Therefore, more than 99.9% of the detected erasable accounts are still stored in the latest StateDB.

Uselessness: In the following, we analyze the detected erasable accounts' transactions to verify their uselessness. All the below analyzed transactions' data is published on https://figshare.com/articles/dataset/11518017.

For the 1,784 DoS contracts, we crawl all of their 26,474 external transactions and 7,707,646 internal transactions. According to the timestamp of Parity multi-sig library's removal (Transaction hash: 0x47f7cff7a5e671884629c93b368cb18f58a99 3f4b19c2a53a8662e3f1482f690), we extract 920 external transactions of attacked Parity wallets that occurred after the attack. Apart from pure ETH transfers, there are 789 external transactions calling wallets' functions. Because calling to a removed contract does not result in failure or exception, we debug these transactions for analysis. We acquire these transactions' execution traces through Geth API `debug_traceTransaction`. All these transactions called the removed library through `DELEGATECALL`, which wasted users' gas or ETH. For malicious DoS contracts, there are 1,128 external transactions used for contract deployments. All the other transactions (15,334 external transactions and 7,700,836 internal transactions) executed with "Out of Gas Error" were exploited for DoS attacks.

For the 479,153 meaningless contracts, we crawl all of their 2,080 external transactions and 490,611 internal transactions for checking and debugging. Apart from 2,002 contract deployment's external transactions, 7 external transactions were halted by the first executed operation STOP before their data fields were processed, which verifies their uselessness. All the other 71 external transactions were executed with "Reverted Error". Apart from 489,890 internal transactions used for contract deployment or compulsive ETH transfer through SELFDESTRUCT, all the other 721 internal transactions were executed with "Reverted Error".

For the 150 stack/opcode error contracts, we crawl all of their 141 external transactions and 157,513 internal transactions for analysis. Apart from 150 transactions used for contract deployment and 337 transactions used for compulsive ETH transfer through SELFDESTRUCT, all the other 157,167 transactions were encountered "Bad Instruction Error" or "Stack Underflow Error".

For the 195 empty accounts, we crawl all of their 237 external transactions and 5 internal transactions. Apart from 195 contract deployment's transactions, we analyze other 47 transactions. All of these 47 transactions transferred ETH or called the empty accounts with function signatures in their data fields, which denotes that they were intended to call a function of contract. However, all of their data fields were not processed because the accounts were empty, which denotes their uselessness. For the 27,200 DoS EOAs, we crawl all of their 1,163,763 internal transactions, and there does not exist external transaction. In 27,200 internal transactions, the attacker created DoS EOA through transferring 1 Wei, which is the smallest financial cost for the attacker. All the other 1,136,563 internal transactions were executed with "Out of Gas Error", which were used for DoS attack (analyzed in Section 4.3.2).

Answer to RQ2 (Accuracy): All the detected erasable accounts' related transactions are useless, and more than 99.9% of the detected erasable accounts are still stored in Ethereum.

4.5.3 RQ3 Waste

In this section, we evaluate the money lost due to erasable accounts. We analyze the gas and ETH consumed in erasable accounts' transactions, whose statistics are shown in Table 4.2. For DoS contracts, 733,583,247 gas were consumed during calling Parity wallets before they were attacked. Therefore, these gas are not wasted. We analyze all the DoS contracts' balance values and 515,035.16 ETH transferred to them are permanently locked in DoS contracts, which are wasted. For meaningless contracts, all their consumed gas are wasted. However, 272.77 ETH attached to their transactions were returned to users due to "Reverted Error", which are not wasted. For category ❸ to ❺, all their gas and ETH are wasted. According to the gas prices set in transactions and ETH price (204.36$/ETH) on May 25 of 2020 [24], 106,360,910$ is totally wasted due to these erasable accounts.

Answer to RQ3 (Waste): About 89 billion gas and 515,037 ETH are wasted due to erasable accounts, which are worth 106,360,910$.

4.6 Graph Analysis

We analyze attacks/behaviors related to discovered erasable accounts to answer the question: **How are erasable accounts behaved and created in reality?**

GLASER's graph analysis module can be divided into two parts, i.e., call graph and creation graph. First, through symbolic execution, we analyze DoS contract's runtime bytecodes to generate call graph from erasable accounts to other accounts. According to the definitions (in Section 4.3) of erasable accounts, only DoS contracts can call other accounts. During symbolic execution, we analyze the operands of DoS related operations (in Section 4.4). If the target address of one operation is a real value, we can conclude that the DoS contract interacts with another account and we add an edge into the call graph. Second, through transaction analysis, we generate creation graph for erasable accounts. We analyze the account creation related transactions of erasable accounts and filter out their source addresses, which constructs the nodes of creation graph. The creation related transactions construct the edges of creation graph. If the erasable account is created through contract account, we also analyze which user (i.e., EOA) calls the contract. Furthermore, we also analyze the creation source address's transactions to see whether it creates other accounts.

Call graph: According to their features, the DoS contracts can be divided into two types, i.e., Many-to-One DoS contract and One-to-Many DoS contract, whose topology graphs are shown in Figure 4.10. We only show the first three bytes of contracts' addresses for better display.

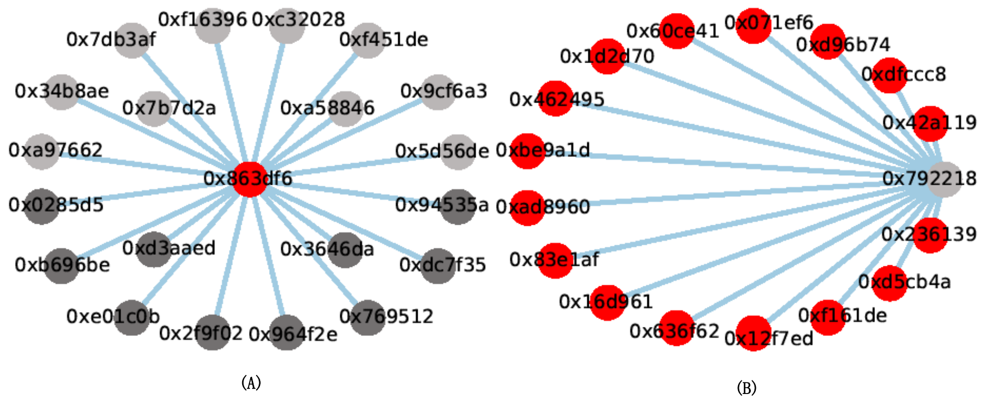


Figure 4.10: (A): Call graph of Many-to-One DoS contracts. (B): Call graph of One-to-Many DoS contract.

For Many-to-One DoS contract, one center contract's address is hardcoded and interacted with many other contracts. Some Many-to-One DoS contracts detected through GLASER are shown in Figure 4.10 (A). In this example, the center contract (Address: 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4) is Parity's multi-sig library, which was attacked in 2017. The center contract is removed, all its dependant wallet contracts become out of service (i.e., dependency error). GLASER has discovered 658 contract accounts calling the removed library. We only show 20 attacked Parity contracts in the figure for better display, and the nodes in deep grey color represent that these erasable contracts still store ETH.

For One-to-Many DoS contract, one DoS contract hardcodes and interacts with many other contracts. One example of One-to-Many DoS contract detected through GLASER is shown in Figure 4.10 (B). In this example, GLASER has discovered that one malicious DoS contract (Address: 0x792218d8bbe00fb81296236b014Fb14af2DA38 5B) hardcodes and interacts with 200 different external contracts, which have all been removed. We only show 16 removed contracts in the figure, and the malicious DoS contract (in light grey) is still stored in StateDB. Both types of DoS contracts might be called, which will result in waste of gas or ETH. For example, one DoS contract (Address: 0x41849f3bd33ced4a21c73fddd4a595e22a3c2251) shown in Figure 4.10 has been transferred ETH in 57 transactions, which can be avoided if its account was detected/alerted in time.

Creation graph: According to their features, the creation graphs can be divided into two types: erasable account created by EOA, and erasable account created by contract.

Erasable accounts were created by EOAs due to programming error or deployment error, and we explain their creations through one meaningless contract example, whose creation graph is shown in Figure 4.11 (A). The user (in red color) called one deployed contract (Address: 0x2Ab748a546760b1EC834E164DEDE2E71C4010E1d) and realized that it was meaningless due to deployment error. The user incorrectly used runtime bytecodes to deploy the contract and transferred ETH to it. Then the user redeployed another correct contract (in green color), whose runtime bytecodes are just same with the data field of the transaction deploying the previous meaningless contract. Note that these types of erasable accounts' creation can be avoided, and it is better to first test and deploy contracts in private/public Testnet before they are deployed in Mainnet.

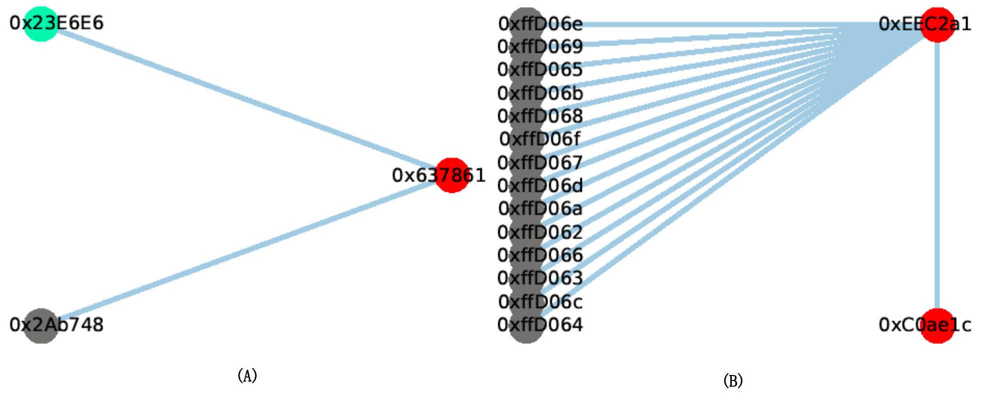


Figure 4.11: (A): Creation graph of one meaningless contract. (B): Creation graph of DoS EOAs.

Erasable accounts were created by contracts due to some attacks, and we explain their creations through one DoS EOA example, whose creation graph is shown in Figure 4.11 (B). The attacker exploited one EOA (Address: 0xc0ae1ca3d89a417cb e525498a1a20d40c9fd720d) to call a malicious contract (Address: 0xeec2a1ee6ee9 42596b6e255d24d38c0a9338cfef), creating 14 different DoS EOAs through internal transactions (Hash: 0xefc6cc36a06eb6b067a35e028a2ad42617d16ff9e958e8fdaec599 d474e306f2). Exploiting one storage variable, the malicious contract can generate different addresses in different transactions. These addresses were calculated and generated in runtime bytecodes, and only the last three bytes of them are different. The attacker totally created 12,204 different DoS EOAs leveraging this malicious contract, resulting in the waste of system resources and nodes' difficulty in syncing data.

4.7 Discussion

We discuss validity threats, limitations, and future work of this chapter.

(1) GLASER can not only discover erasable accounts that already exist in Ethereum, but also erasable accounts that might be created in future. Some kinds of accounts analyzed by GLASER might also be created in future, and GLASER might discover more erasable accounts.

(2) For the discovered erasable accounts, only part of meaningless contracts can be destructed by ordinary users. Because some MC-RS have `SELFDESTRUCT` in their first basic blocks, which can be invoked through transactions by users. Although most of discovered erasable accounts cannot be easily destructed by users, our results can remind users not to call them, which can help users save money.

(3) Path explosion and timeout exception are common threats for the symbolic execution techniques leveraged in this chapter. However, we use some methods to reduce these threats. During detecting stack error contracts, we first extract runtime bytecodes corresponding to the first basic block and then symbolically execute them. During detecting attacked Parity wallets, we first filter out contracts without external call operations and then symbolically execute them.

(4) As GLASER focuses on five kinds of erasable accounts in Ethereum, we will

detect more kinds of erasable accounts in future. We will also analyze erasable accounts in other blockchain systems.

4.8 Brief Summary

We have conducted the first work that systematically characterizes erasable accounts in Ethereum, i.e., erasable contract accounts and erasable EOAs. We have implemented GLASER to analyze the StateDB, which can detect erasable accounts leveraging bytecodes' static analysis, symbolic execution, transaction analysis, and state fields analysis. Furthermore, we have analyzed attacks/behaviors related to erasable accounts through graph analysis. Extensive experiments are also conducted to evaluate the quantity, accuracy, and waste of the detected erasable accounts.

Chapter 5

Gas-Inefficient Contracts and Locked Cryptocurrencies that Devour Money

Smart contracts are full-fledged programs that run on blockchains (e.g., Ethereum, one of the most popular blockchains). In Ethereum, gas (in Ether, a cryptographic currency like Bitcoin) is the execution fee compensating the computing resources of miners for running smart contracts. However, we find that under-optimized smart contracts cost more gas than necessary, and therefore the creators or users will be overcharged. In this work, we conduct the *first* investigation on Solidity, the recommended compiler, and reveal that it fails to optimize gas-inefficient programming patterns. In particular, we identify seven gas-inefficient patterns and group them to two categories. Then, we propose and develop Gasper, a new tool for automatically locating gas-inefficient patterns by analyzing smart contracts' bytecodes. The preliminary results on discovering three representative patterns from 4,240 real smart contracts show that 93.5%, 90.1% and 80% contracts suffer from these three patterns, respectively.

As the most popular blockchain that supports smart contracts, there are already more than 296 thousand kinds of cryptocurrencies built on Ethereum. However, not all cryptocurrencies can be controlled by users. For example, some money is permanently locked in wallets' accounts due to attacks. In this chapter, we conduct the *first* systematic investigation on locked cryptocurrencies in Ethereum. In particular, we define four categories of accounts with locked cryptocurrencies and develop a novel tool named CLUE to discover them. Results show that there are more than one billion dollars value of cryptocurrencies locked in Ethereum. We also analyze the reasons (i.e., attacks/behaviors) why cryptocurrencies are locked. Because the locked cryptocurrencies can never be controlled by users, avoid interacting with the accounts discovered by CLUE and repeating the same mistakes again can help users to save money.

5.1 Overview

Smart contracts run on the machines of miners, who can earn Ethers (i.e., the cryptographic currency circulated in Ethereum) by contributing their computing resources. The creators and users of smart contracts will be charged certain amount of gas for purchasing the computing resources from miners. The charge of a transaction equals to the multiplication of the gas consumed by executing the transaction and the price of gas (ETH per unit). Moreover, when deploying contracts, the creators will also be charged of gas, the amount of which are related to the size of smart contracts in bytecodes.

We find that under-optimized smart contracts cost more gas than necessary, and therefore the creators or users will be overcharged. To save money, developers had better follow gas-efficient programming patterns. Unfortunately, there is not such a guideline yet, and it is difficult for developers to identify gas-inefficient bytecode and replace them with gas-efficient ones, because it requires deep understanding of EVM's instructions, the gas consumption for different operations, the data locations accessed by operations, the amount of data read or written etc. Hence, a compiler that can optimize the bytecode for minimizing gas consumption is highly desired.

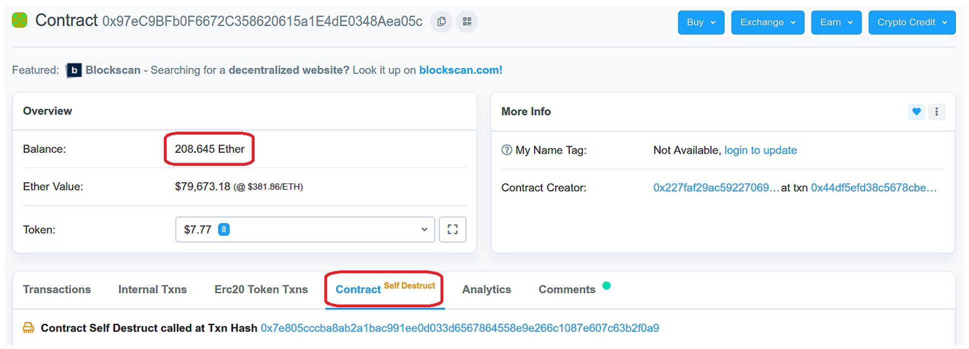
In this chapter, we conduct the *first* investigation on Solidity, the recommended compiler for Ethereum, and reveal that it fails to optimize gas-inefficient programming patterns. More precisely, we identify seven gas-inefficient patterns and divide them into two categories: useless-code related patterns, and loop-related patterns. Furthermore, we propose and develop GASPER (short for GAS-inefficient Patterns checkER), a new tool for discovering gas-inefficient patterns in bytecode automatically. GASPER leverages symbolic execution and it currently can locate three representative patterns, which cover the two categories. By applying GASPER to analyze all deployed smart contracts, we find that 93.5%, 90.1% and 80% smart contracts suffer from these three patterns, respectively. It is worth noting that although the list of our patterns is by no means of complete, this research sheds light on this important issue and hopefully stirs more research on it.

As the most popular blockchain that supports smart contracts, there are many kinds of contract-based cryptocurrencies built in Ethereum. Apart from ETH, which is the native cryptocurrency of Ethereum, more than 296 thousand cryptocurrency contracts are deployed in Ethereum [54]. These cryptocurrencies have high market capitalization. For example, the ETH has a total value of about 20 billion dollars [56], and USDT has a total value of more than four billion dollars [53]. Note that all the cryptocurrencies' prices in this chapter are based on statistics in September, 2020 [54].

However, not all cryptocurrencies can be controlled by users. Actually, much value of cryptocurrency is permanently locked in some accounts. For example, the attacker escalated his privilege and destructed Parity's multi-sig library contract in 2017 [45], which locked all the ETH stored in Parity wallet accounts. Through our analysis, there are 203 wallet accounts with more than 515,035 locked ETH, which is worth more than 192 million dollars. Many users still sent cryptocurrencies to the attacked wallet accounts, leading more money permanently lost. If the accounts with locked cryptocurrencies can be detected and alerted in time, users can reduce their economic losses.

Unfortunately, there still lacks systematic research on the locked cryptocurrencies in Ethereum. Although there are some research analyzing attacks [87, 118] or criminal smart contracts [86] stealing cryptocurrencies in Ethereum, they focus on different purposes from ours. We detect locked cryptocurrencies that do not belong to anyone, while they analyze stolen cryptocurrencies due to attacks. There also exist some other research analyzing the properties of cryptocurrency networks [153, 165], inconsistent behaviors [83], and gas optimizations [81] of smart contract in Ethereum, whose main contents and purposes are different from this chapter.

To fill this gap, we propose and develop a novel tool named CLUE (disCovering Locked cryptocUrrency in Ethereum), which can discover four categories of accounts with more than one billion dollars value of locked cryptocurrencies. In particular, we discover two categories of contract accounts with locked cryptocurrencies due to contract destruction or attacks, and two categories of EOAs (Externally Owned Accounts) with locked cryptocurrencies due to users' unreasonable behaviors. Note that calling to accounts with locked cryptocurrencies not only wastes system compu-


Contract 0x97eC9BFb0F6672C358620615a1E4dE0348Aea05c

Featured: Blockscan - Searching for a **decentralized website?** Look it up on **blockscan.com!**

Overview	
Balance:	208.645 Ether
Ether Value:	$79,673.18 (@ $381.86/ETH)
Token:	$7.77 8

More Info	
My Name Tag:	Not Available, login to update
Contract Creator:	0x227faf29ac59227069...at txn 0x44df5efd38c5678cbe...

Contract Self Destruct called at Txn Hash 0x7e805cccba8ab2a1bac991ee0d033d6567864558e9e266c1087e607c63b2f0a9

Figure 5.1: One destructed contract account with locked cryptocurrency detected by CLUE.

tation resources, but also wastes users' money. For example, one contract account (Address: 0x97eC9BFb0F6672C358620615a1E4dE0348Aea05c) discovered by CLUE locks more than 208 ETH, as shown in Figure 5.1, which is worth more than 79 thousand dollars. Users still call this contract after its destruction, leading to all the sent cryptocurrencies locked permanently.

Our Contributions. The main contributions of this chapter are listed as follows:

- To our best knowledge, this is the *first* investigation revealing that lots of smart contracts, generated by the recommended compiler, contain gas-inefficient bytecodes, which can be replaced with gas-efficient bytecodes to save money.

- We propose and develop GASPER, a new tool based on symbolic execution for automatically discovering gas-inefficient patterns in bytecode. The current version covers three representative patterns in two categories, and is being extended to support more patterns.

- We apply GASPER to all deployed smart contracts, and find that 93.5%, 90.1% and 80% smart contracts suffer from these three patterns, respectively.

- To the best of our knowledge, we conduct the *first* research that systematically analyzes locked cryptocurrencies in Ethereum. We propose and define four categories of accounts with locked cryptocurrencies, i.e., two kinds of EOAs and two kinds of smart contract accounts.

- We implement a tool named Clue to detect each category of accounts with locked cryptocurrencies. For smart contract accounts, we analyze their account states in StateDB and analyze their historical transactions, to discover destructed contracts. Leveraging symbolic execution, we analyze the runtime bytecodes of smart contracts to discover attacked Parity wallet contracts. For EOAs, we mainly use account state analysis and transaction analysis to detect contract-creation failure EOAs and `0x0` account.

- We analyze the attacks/behaviors related to the discovered locked cryptocurrencies, which can explain why they are locked and help users to save money. We also conduct experiments to evaluate its quantity and accuracy. A total of 1,091,796,292.09$ value of cryptocurrencies are discovered by Clue, and all of the discovered cryptocurrencies are permanently locked in Ethereum.

This chapter is organised as follows. Section 5.2 introduces the related technical background. Section 5.3 interprets the details of gas-inefficient programming patterns and Section 5.4 details the principles of Gasper. Then Section 5.5 evaluates Gasper's results. Section 5.6 describes the four categories of accounts with locked cryptocurrencies and Section 5.7 details the principle and implementation of Clue. Then Section 5.8 systematically evaluates the quantity and accuracy of discovered locked cryptocurrencies. After discussing the limitations and future work in Section 5.9, we summarize the chapter in Section 5.10.

5.2 Background

In this section, we briefly introduce the necessary background related to this chapter.

5.2.1 Gas Mechanism

Gas is used for purchasing computing resources from miners since smart contracts run on miners' machines. Gas can be considered as money with equivalent value. For example, the average gas price on Nov. 11, 2016 is 0.000000024334480804 ETH [26], which is roughly equal to 2.5×10^{-7} US dollars [9]. Note that the gas price and the exchange rate of ETH to US dollar are determined by the market and keep changing.

Deploying and executing smart contracts cost money. For instance, an addition operation that sums up the top two items of the stack takes 3 units of gas, about 7.5×10^{-7} US dollars. One may argue that the cost for an addition is so low that we do not need to optimize it. However, it is worth noting that real smart contracts consist of lots of operations and some operations consume much more gas than the addition operation, as shown in Table 5.1. Moreover, smart contracts usually provide public methods that can be called unlimited times by various clients and contracts. Hence, an optimized smart contract can save obvious gas (i.e., money) than its un-optimized counterpart due to the scale effect.

Stack operations (e.g., `POP`, `PUSH`), arithmetic operations (e.g., `ADD`, `SUB`), bitwise operations (e.g., `OR`, `XOR`), and comparison operations (e.g., `LT`/`GT`) are cheap because being a stack-based virtual machine, EVM favors such stack-related operations. Loading a word (i.e., 256 bits) from the memory (e.g., `MLOAD`) or saving a word to

Table 5.1: Gas cost of different operations, a complete list can be found in Ethereum's yellow paper.

Operation	**Gas**	**Description**
ADD/SUB	3	Arithmetic operation
MUL/DIV	5	
ADDMOD/MULMOD	8	
AND/OR/XOR	3	Bitwise logic operation
LT/GT/SLT/SGT/EQ	3	Comparison operation
POP	2	Stack operation
PUSH/DUP/SWAP	3	
MLOAD/MSTORE	3	Memory operation
JUMP	8	Unconditional jump
JUMPI	10	Conditional jump
SLOAD	200	Storage operation
SSTORE	5,000/ 20,000	
BALANCE	400	Get balance of an account
CREATE	32,000	Create a new account using CREATE
CALL	25,000	Create a new account using CALL

the memory (e.g., `MSTORE`) are also cheap. The term "memory" referred in Ethereum stands for a special memory area, of which a contract obtains a freshly cleared instance for each message call. For example, the data attached in a message call is stored in memory. It is worth noting that the gas consumption will be multiplied if many words in memory are read or written. Moreover, memory can be expanded when accessing a previously untouched memory location. Every expanded word needs 3 units of gas.

Loading a word from the storage (i.e., `SLOAD`) or saving a word to the storage (i.e., `SSTORE`) are expensive. The term "storage" referred in Ethereum is a persistent memory area where any changes to the storage by one call of a contract can be observed by subsequent calls of that contract. A `SSTORE` operation costs 20,000 units of gas if the storage word is set to non-zero from zero; otherwise, it costs 5,000. It is worth noting that although the caller of a contract will be refunded 15,000 units of gas if a `SSTORE` operation sets a non-zero storage word to zero, the refund will not be committed until the transaction completes successfully.

EVM has a number of blockchain-specific operations which are very expensive, such as `BALANCE`, `CREATE` and `CALL`. Moreover, a conditional jump (i.e, `JUMPI`) is more expensive that an unconditional jump (i.e., `JUMP`). The gas consumption of each operation is susceptible to change due to the fast evolving of Ethereum. Roughly speaking, users are charged proportionally to the consumed computing resources.

5.2.2 Account State

Every account in Ethereum has its state, whose information is stored in StateDB (State DataBase). For each account a, there are four fields in StateDB [130]:

(1) Code $\sigma[a]_c$. It stores the smart contract's runtime bytecodes, which is empty if a is an EOA.

(2) Balance $\sigma[a]_b$. It stores the ETH balance value (in Wei) of the account. 1 ETH = 10^{18} Wei.

(3) Nonce $\sigma[a]_n$. It stores the number of transactions *sent from* EOA, or the number of contracts created by contract account.

(4) Storage $\sigma[a]_s$. It stores the root hash of Merkle tree which encodes the storage contents of the account.

The main difference of EOA and contract account is whether its code field $\sigma[a]_c$ is empty [130]. In this chapter, we mainly analyze three fields of account, i.e., $\sigma[a]_c$, $\sigma[a]_b$, and $\sigma[a]_n$.

5.2.3 Cryptocurrencies in Ethereum

Cryptocurrency is digital assets based on blockchain techniques [153]. There are two categories of cryptocurrencies in Ethereum, i.e., ETH and CBC (Contract-Based Cryptocurrency) [165]. ETH is the native cryptocurrency of Ethereum, and miners can earn ETH by participating in consensus and writing blocks. Apart from ETH, there are many other kinds of cryptocurrencies based on contracts, and ERC20 is the most popular standard of CBC [131]. All the CBC analyzed in this chapter are compliant with ERC20. Both EOA and contract account can hold cryptocurrency. EOA can transfer out ETH by initiating transactions from it, and contract can transfer out ETH by executing specific operations (e.g., `CALL`, `SELFDESTRUCT`) [62]. Note that accounts can only transfer out their CBC by calling the corresponding ERC20-based smart contract.

The ERC20 standard provides some basic functions and events that must be implemented of CBC in Ethereum, which are shown in Figure 5.2. The constant string *name* defines the name of the CBC (e.g., USDT). If the user U_a wants to transfer out CBC from his/her account, U_a can call the function `transfer()` (in Line 5). Furthermore, the user U_a can authorize another account U_b to transfer out CBC from his/her account through calling the function `transferFrom()` (in Line 6). Before U_b transfers out U_a's CBC, U_a must authorize the account U_b through calling function `approve()` (in Line 7). In the bodies of function `transfer()` and `transferFrom()`, developers need to call the event `Transfer()` (in Line 9), because Etherscan needs to monitor the CBC's transfer information through this event [153, 165].

```
contract ERC20 {
  string public constant name = "Token Name";
  function totalSupply() constant returns (uint sup);
  function balanceOf(address owner) constant returns (uint balance);
  function transfer(address to, uint value) returns (bool success);
  function transferFrom(address from, address to, uint value) returns (bool success);
  function approve(address spender, uint value) returns (bool success);
  function allowance(address owner, address spender) constant returns (uint remaining);
  event Transfer(address indexed from, address indexed to, uint value);
  event Approval(address indexed owner, address indexed spender, uint value);  }
```

Figure 5.2: The ERC20 standard of CBC in Ethereum.

5.2.4 Cryptocurrency Analysis

Chen et al. [86] detected Ponzi schemes, which are classic frauds and might cheat users' ETH. They built a classification model to detect latent Ponzi schemes by using data mining and machine learning methods. Cheng et al. [87] analyzed the attack that steals cryptocurrencies exploiting unprotected JSON-RPC endpoints. They designed and implemented a honeypot that could capture real attacks in the wild. Ji et al. [118] implemented a tool named DEPOSAFE to detect and exploit the fake deposit vulnerability in ERC-20 tokens. They identified over 7,000 vulnerable contracts that may suffer from two types of attacks, which demonstrated the urgency to identify and prevent the fake deposit vulnerability. However, all of the above work analyzed the cryptocurrencies illegally possessed by criminals, and they did not analyze locked cryptocurrencies that does not belong to anyone. [153, 154, 165] measured the network properties and structures of ERC20 smart contracts, and [83] analyzed inconsistent behaviors in ERC20 smart contracts. However, they focused on analyzing the smart contracts' implementations and invocations, whose purposes differ from ours. Angelo et al. [95] detected and classified the bytecodes of cryptocurrency related smart contracts. They analyzed the methods and standards of the contracts, to investigate the actual usage of cryptocurrency related contracts. However, they did not analyze the states/transfers of cryptocurrencies held by accounts and did not analyze locked cryptocurrencies. Tramèr et al. [161] analyzed the side-channel attacks that let remote adversaries bypass privacy protections in cryptocurrency related transactions. Their proposed attacks enabled an active remote adversary to identify the secret payee of any transaction in Zcash or Monero. Although their study highlighted the dangers of side-channel leakage in anonymous cryptocurrencies, they did not analyze locked cryptocurrencies in these systems. Zhou et al. [181] analyzed real-world attacks and defenses adopted in the wild based on the transaction logs produced by uninstrumented EVM. Besides successful attacks, they also studied attempted attacks that are prevented due to the deployments of defenses. Although they analyzed cryptocurrency transfer actions resulted from the attacks, they did not analyze the locked cryptocurrencies due to attacks or unreasonable behaviors. Kalodner et al. [120] presented BLOCKSCI, a platform to analyze the data of different blockchain systems. BLOCKSCI incorporated an in-memory analytical (rather than transactional) database and it supported different analysis tasks, including cryptocurrency analysis. However, their study focused on the acquisition and organization of blockchain data and they did not analyze locked cryptocurrencies. There are some other work analyzed cryptocurrencies in Ethereum [101, 104, 148, 155] or other blockchains [72, 93, 114, 174], whose contents and purposes are different from ours.

5.2.5 Gas-Related Studies

Chen et al. propose to defend against DoS attacks of Ethereum by adjusting gas costs dynamically [81]. By doing so, DoS attacks exploiting under-priced EVM operations will be terminated quickly [81]. MADMAX decompiles EVM bytecode, and then detects gas-related security problems from the decompiled code [108]. GASTAP derives the sound gas upper bounds for all public functions of a given smart contract, by inferring size relations, generating gas equations and solving the equations [66]. GASOL extends GASTAP by replacing multiple accesses to the same storage location with one access [64]. GASFUZZ applies feedback-directed fuzz testing to automatically generate inputs which could lead to a high gas consumption of contract functions [137]. Marescotti et al. [138] leverages symbolic model checking to compute the exact worst-case gas consumption for smart contracts.

Yang et al. conduct an empirical study of gas consumption, and they have several observations, including some under-priced EVM operations that can be exploited by DoS attacks [170]. Zhang et al. propose a novel data structure, so-called GEM2-Tree to substitute the original Merkle hash tree in Ethereum to reduce gas cost [175]. SMARTCHECK [160] detects 21 kinds of code issues in Ethereum smart contracts. Two of them are related to gas-inefficient code. The first is byte[] because it can be replaced with bytes which is cheaper. The second is the loops with function calls inside because repeated function invocations will result in considerable gas consumption. Our work is different with SMARTCHECK mainly because SMARTCHECK relies on the source code of smart contracts. Unfortunately, open-source contracts only account for less than 1%. For example, since there is no type information in EVM bytecode, it is difficult to distinguish byte[] from bytes without the source code.

5.3 Gas-Inefficient Programming Patterns

We identify seven gas-inefficient patterns, which can be classified into two categories: useless code related patterns and loop related patterns. The former introduces additional cost due to the increased size of bytecode during the deployment and the removable bytecode in runtime. The latter involves using expensive operations in the loop. We have validated all these patterns using the latest Solidity whose optimization is enabled. More precisely, we feed Solidity the gas-inefficient patterns in source code, and then check whether the gas-inefficient patterns are converted into gas-efficient ones in the generated bytecode. The results show that none of these patterns has been optimized by Solidity. For the ease of illustration, we present the patterns in source code rather than bytecode.

5.3.1 Category 1: Useless Code Related Patterns

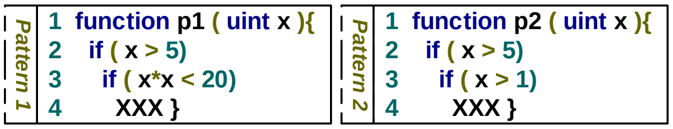

Pattern 1
```
1  function p1 ( uint x ){
2    if ( x > 5)
3      if ( x*x < 20)
4        XXX }
```
Pattern 2
```
1  function p2 ( uint x ){
2    if ( x > 5)
3      if ( x > 1)
4        XXX }
```

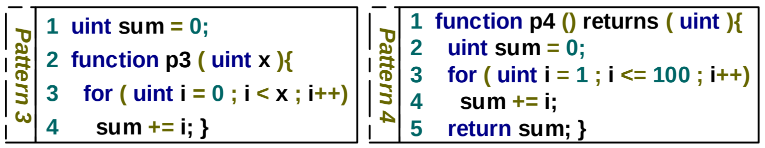
Figure 5.3: Pattern 1: dead code, and Pattern 2: opaque predicate

(1) Dead code. Figure 5.3 (Pattern 1) gives an example of dead code where Line 4 will not be executed because the predicate “x*x<20” at Line 3 is evaluated to false

under all circumstances. Solidity does not remove Line 3 and 4 from the generated bytecode and hence wastes money.

(2) Opaque predicate. The outcome of an opaque predicate is known to be true or false without execution. For example, the predicate “x>1” in Figure 5.3 (Pattern 2) is an opaque predicate. Since the predicate at Line 3 is evaluated to true under all circumstances, it should be removed for saving gas.

5.3.2 Category 2: Loop Related Patterns



Pattern 3

```
1 uint sum = 0;
2 function p3 ( uint x ){
3   for ( uint i = 0 ; i < x ; i++)
4     sum += i; }
```

Pattern 4

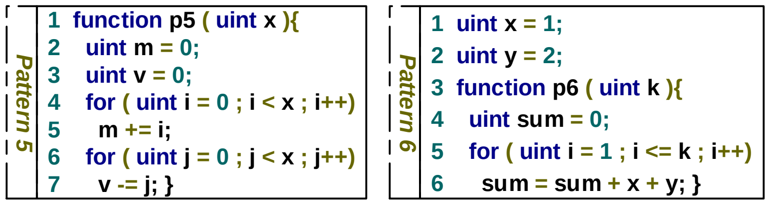
```
1 function p4 () returns ( uint ){
2   uint sum = 0;
3   for ( uint i = 1 ; i <= 100 ; i++)
4     sum += i;
5   return sum; }
```

Figure 5.4: Pattern 3: expensive operations in a loop, and Pattern 4: constant outcome of a loop

(1) Expensive operations in a loop. The expensive operations in a loop are worth attention because they may execute multiple times in one invocation. Moving the expensive operations out of the loop can save gas. For example, in Figure 5.4 (Pattern 3), since the variable *sum* is stored in the storage, Line 4 involves a `SLOAD` for loading *sum* to the stack and a `SSTORE` for saving the outcome of the `ADD` to the storage. Note that the storage-related operations are very expensive.

An advanced compiler should assign *sum* to a local variable (e.g., *tmp*) that resides in the stack, then add i to *tmp* inside the loop, and finally assign *tmp* to *sum* after the loop. Such optimization reduces the storage-related operations from $2x$ to just 2, i.e., one `SLOAD` and one `SSTORE`.

(2) Constant outcome of a loop. In some cases, the outcome of a loop may be a constant that can be inferred in compilation. As shown in Figure 5.4 (Pattern 4), the storage variable *sum* in *p*4 equals to 5050 after the loop. Hence, the body of *p*4 should be simplified as “return 5050;”.



```
1 function p5 ( uint x ){
2   uint m = 0;
3   uint v = 0;
4   for ( uint i = 0 ; i < x ; i++)
5     m += i;
6   for ( uint j = 0 ; j < x ; j++)
7     v -= j; }
```

```
1 uint x = 1;
2 uint y = 2;
3 function p6 ( uint k ){
4   uint sum = 0;
5   for ( uint i = 1 ; i <= k ; i++)
6     sum = sum + x + y; }
```

Figure 5.5: Pattern 5: loop fusion, and Pattern 6: repeated computations in a loop

(3) Loop fusion. It combines several loops into one if possible and thus reduces the size of bytecode. In particular, it can reduce the amount of operations, such as conditional jumps and comparison, etc., at the entry points of loops. The two loops shown in Figure 5.5 (Pattern 5) can be combined into one loop, where both *m* and *v* get updated.

(4) Repeated computations in a loop. In some cases, there may be expressions that produce the same outcome in each iteration of a loop. Hence, the gas can be saved by computing the outcome once and then reusing the value instead of recomputing it in subsequent iterations, especially, for the expressions involving expensive operands. For example, in Figure 5.5 (Pattern 6), the gas consumption is very high due to the repeated computations. More precisely, the summation of two storage words (i.e., “x+y” at Line 6) is quite expensive because *x* and *y* should be loaded into the stack (i.e., `SLOAD`) before addition. To save gas, this summation should be finished before the loop, and then the result is reused within the loop.

Pattern 7

```
1  function p7 ( uint x , uint y ) returns ( uint ){
2    for ( int i = 0 ; i < 100 ; i++)
3      if ( x > 0 )  y+=x;
4    return y; }
```

Figure 5.6: Pattern 7: Comparison with unilateral outcome in a loop

(5) Comparison with unilateral outcome in a loop. It means that a comparison is executed in each iteration of a loop but the result of the comparison is the same even if it cannot be determined in compilation (i.e., not an opaque predicate). For instance, in Figure 5.6, the comparison at Line 3 should be moved to the place before the loop.

Summary: Adequate optimizations can reduce the cost of contract creators if the size of smart contracts can be reduced (e.g., eliminating dead code, removing unnecessary comparisons), and the cost of contract users if the computations of smart contracts can be reduced (e.g., moving expensive operations out of a loop). It is worth noting that the loop-related patterns will cost more gas with the increase of the loop count.

5.4 GASPER's Implementation

We propose and develop GASPER to automatically discover gas-inefficient programming patterns from the bytecode of smart contracts. GASPER handles bytecode directly without the need of source code, because only a few smart contracts open their sources. As an early research achievement, the current version of GASPER can find all patterns in category 1 and one representative pattern (i.e., expensive operations in a loop) in category 2. The detection of other patterns is in development.

GASPER conducts symbolic execution on bytecode to cover all reachable code blocks (a block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit). Given a smart contract, GASPER first disassembles its bytecode using *disasm* provided by Ethereum. Then, GASPER constructs the Control Flow Graph (CFG). It is worth noting that the CFG will be improved gradually during symbolic execution if new control flow transfers are found. Symbolic execution starts from the root node of the CFG, and traverses the CFG. If GASPER encounters a conditional jump, it checks which branches (i.e., true or false) are feasible by querying the Z3 solver [92]. If both are feasible, GASPER selects one branch following the depth-first search.

5.4.1 Detection of Dead Code

GASPER detects dead code through three steps. First, it logs the addresses of all executed blocks by symbolic execution. Then, it collects the addresses of all blocks by scanning the CFG. Finally, GASPER reports all blocks that are found in the CFG but not executed by symbolic execution as dead code.

5.4.2 Detection of Opaque Predicates

To detect opaque predicates, GASPER executes the smart contract symbolically, and records the executed branch (i.e., true or false) when a conditional jump is encountered. After that, the conditional jump with one never-executed branch is regarded as an opaque predicate.

5.4.3 Detection of Expensive Operations in a Loop

Gasper detects this pattern through two steps. First, Gasper looks for loops in the bytecode. Second, it searches loop bodies for expensive operations. More precisely, Gasper firstly searches for back edges in the CFG, which indicate the existence of loops, and then identifies the entry block and exit block for each loop. Afterwards, using Dijkstra algorithm, Gasper calculates the distances between each block with the entry block and exit block, respectively. The distance between two nodes is the least number of edges from one node to the other. A block is considered to be in a loop if it is closer to the exit block than to the entry block. Currently, Gasper supports detecting 3 expensive operations, including `SLOAD`, `SSTORE` and `BALANCE`. More operations will be included in future work.

5.5 Gasper's Evaluation

We have implemented Gasper based on Oyente [134], and evaluated it using all smart contracts deployed on Ethereum. More precisely, we scan all addresses in the blockchain because each deployed contract must be associated with an unique address. We find 566,907 addresses till November 5th, 2016, of which 539,617 addresses contain no bytecodes. Therefore, we download 27,290 contracts' bytecodes in total. Moreover, we find that many contracts are exactly the same (i.e., their bytecodes are identical). After eliminating identical contracts, 4,669 contracts are left. During experiments, 429 (less than 10%) contracts cannot be examined because Oyente crashes due to its internal errors(e.g., Unknown Instructiondelegatecall, Stack Underflow, Unknown Instructionextcodesize) or Oyente runs out of time. Eventually, 4,240 contracts are

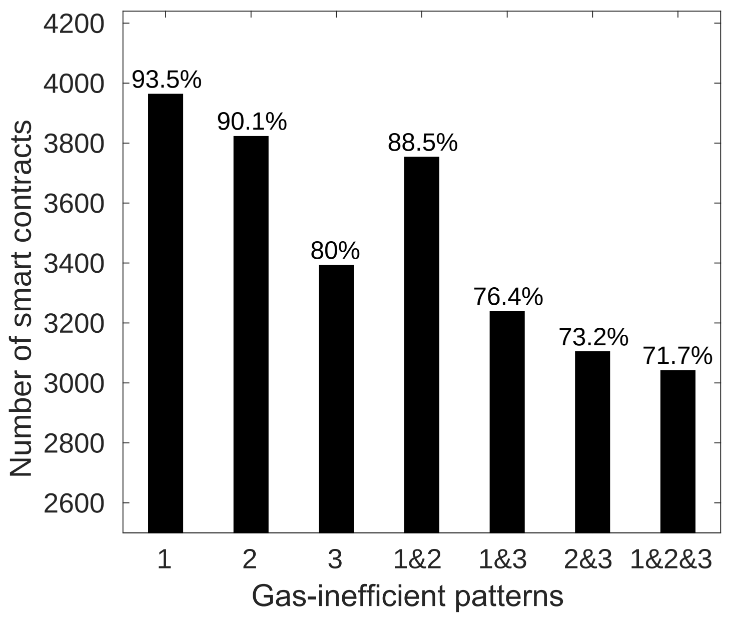


Figure 5.7: Overview of gas-inefficient patterns: 1, 2, 3 indicate dead code, opaque predicates, and expensive operations in a loop, respectively.

successfully inspected.

The number of smart contracts that have the 3 gas-inefficient patterns are illustrated in Figure 5.7. More than 70% contracts contain all these patterns, indicating that their bytecodes have not been properly optimized for reducing gas. Besides, more than 90% contracts have dead code or opaque predicates.

Figure 5.8 presents the distribution of dead code blocks and opaque predicates in smart contracts. Each point (a, b) indicates that a smart contracts contain b dead code blocks or opaque predicates. Note that the contracts without these two patterns are not counted. The distributions of dead code blocks and opaque predicates demonstrate similar trends: 51.7% contracts contain more than 20 dead code blocks and 52.6% contracts contain more than 10 opaque predicates.

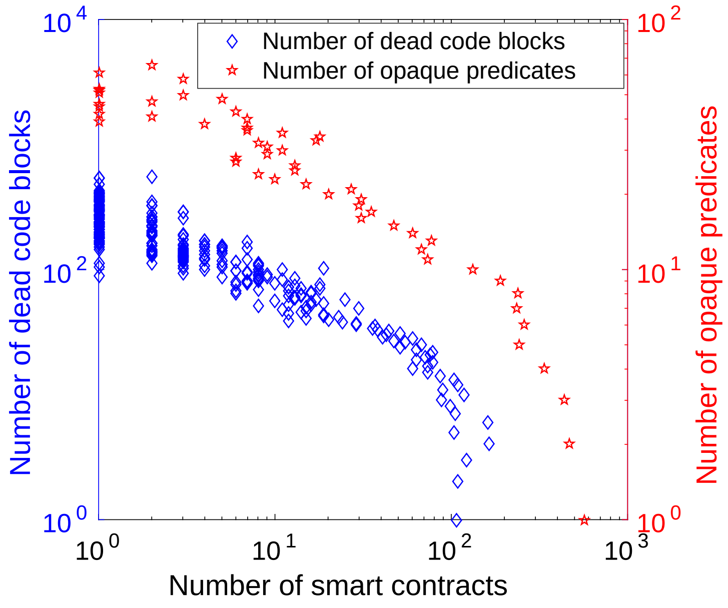


Figure 5.8: Distribution of dead code blocks and opaque predicates in smart contracts.

Figure 5.9 demonstrates that 69.9%, 78.5% and 21% contracts have `SLOAD`, `SSTORE` and `BALANCE` operations in a loop, respectively. Moreover, if a contract has `SSTORE` operations in a loop (the percentage is 69.9%), it may contain `SLOAD` operations (69.3%) as well. Interestingly, if a contract uses `BALANCE` operations in a loop (21%), it likely contains both `SLOAD` and `SSTORE` operations (18.6%).

Figure 5.10 shows that a large number of contracts contain many expensive operations in a loop. For example, 57.1% and 51.5% of contracts have more than 7 `SSTORE` and 20 `SLOAD` operations in a loop, respectively. Note that contracts without such expensive operations in a loop are not counted.

As expected, contracts with larger size are likely to contain more gas-inefficient patterns. Figure 5.11 shows the relationship between the number of `SLOAD/SSOTRE` and the size of smart contracts. For example, a contract, named *ARK*, which is of

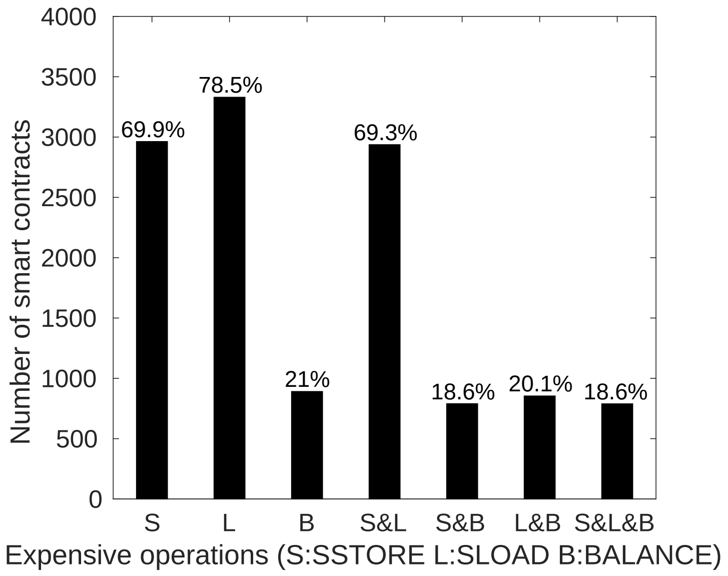


Figure 5.9: Number of contracts containing expensive operations

34,767 bytes and deployed in 0x37b4869e73B7cE1284D6502B01aC81d500b50237, has 304 `SLOAD` and 168 `SSTORE` operations in loops.

5.5.1 Real Case 1: FIRSTCONTRACT

FIRSTCONTRACT is open source and deployed at the address 0x68C7147205A8bEB9 D99fD19908b93462CdFfC60d. GASPER discovers dead code at Line 200 (i.e., pattern 1) and an opaque predicate (i.e., pattern 2) at Line 199, as shown in Figure 5.12. The function *indexof* takes in two strings, *_haystack* and *_needle*. At Line 195, *_haystack* is converted into a set of bytes, *h*. At Line 199, the length of *h* is compared to $2 * *128 - 1$. However, the predicate will never be evaluated to true because "∗∗" stands for exponential arithmetic. Consequently, the code at Line 200 cannot be executed.

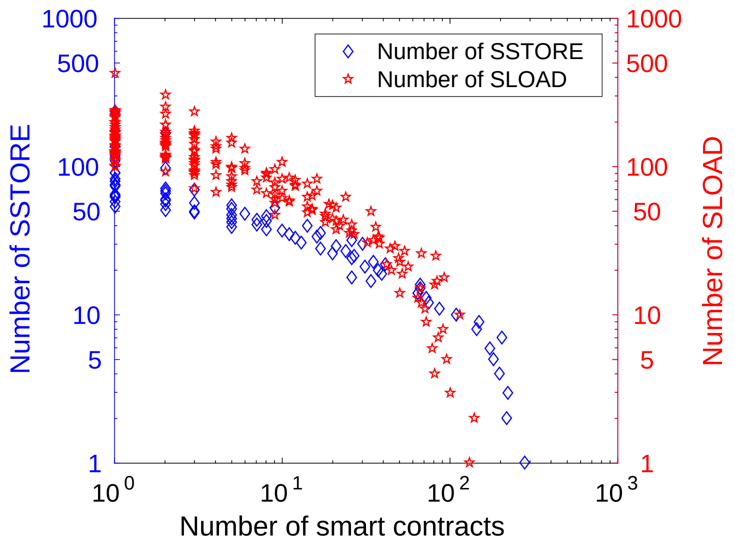


Figure 5.10: Distribution of `SSTORE` and `SLOAD` within a loop in smart contracts.

5.5.2 Real Case 2: Ballot

Ballot is also open source and deployed at the address 0x5A4964bb5FDd3CE646b B6AA020704F7D4db79302. Gasper finds a `SLOAD` operation in a loop and it can be moved outside the loop, as shown in Figure 5.13.

Since the array *proposals* (defined Line 29) is in the storage, getting access to its length (i.e., *proposals.length* at Line 59) involves the `SLOAD` operation. Moreover, the number of executing `SLOAD` is *proposals.length*, because the length of *proposals* is accessed in each iteration of the loop. This inefficient code can be optimized by assigning *proposals.length* to a stack variable, and then using the stack variable to do the comparison with *proposal* at Line 59. After optimization, the number of using `SLOAD` can be reduced to only one.

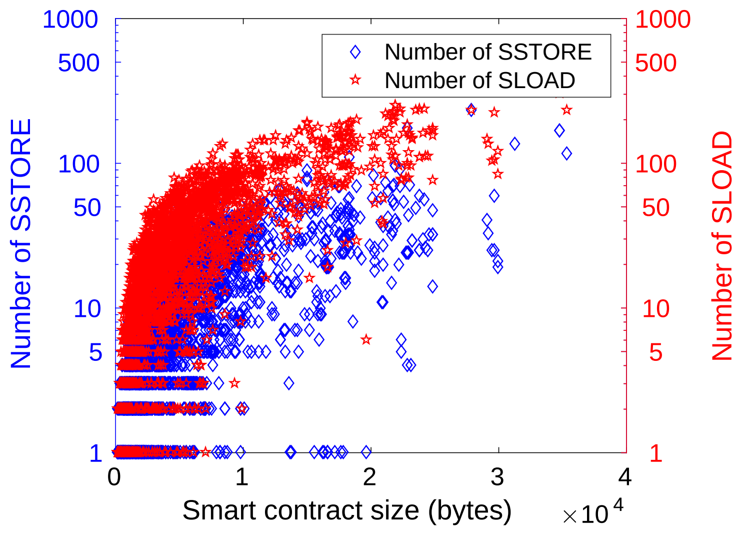


Figure 5.11: Statistics of the size of contracts which contain `SSTORE` and `SLOAD` in a loop.

```
function indexOf ( string _haystack , string _needle ) internal returns ( int )
{
  bytes memory h = bytes ( _haystack );
  bytes memory n = bytes ( _needle );
  if ( h . length < 1 || n . length < 1 || ( n . length > h . length ))
    return - 1;
  else if ( h . length > ( 2 ** 128 - 1 ))
    return - 1 ;
 ...
```

Figure 5.12: Gas-inefficient code in FIRSTCONTRACT

5.6 Locked Cryptocurrencies

In this section, we define four kinds of contract accounts and EOAs with locked cryptocurrencies in Ethereum.

```
Proposal [] public proposals;
 ...
function winningProposal () constant returns ( uint8 winningProposal){
  uint256 winningVoteCount = 0 ;
  for ( uint8 proposal = 0 ; proposal < proposals . length ; proposal ++)
 ...
```

Figure 5.13: Gas-inefficient code in BALLOT

5.6.1 Locked Cryptocurrencies in Contract

Destructed contract

In Ethereum, the smart contract can be destructed and transfer out all its stored ETH through executing the `SELFDESTRUCT` operation. After destruction, the smart contract account will be deleted from StateDB. However, some users may not know in time of the smart contract's destruction and still send ETH/CBC to it, which leads to the sent ETH/CBC be locked. After sending ETH to the destructed smart contract account, the contract account with the same address before destruction will be created again in the StateDB. For the CBC held by the destructed contract account, most of it will also be permanently locked. Because the destructed contract account stores no runtime bytecodes, it cannot send out transaction. Therefore, the destructed contract account cannot transfer out its CBC through calling `transfer()` function, or authorize another account to transfer out its CBC through calling the `transferFrom()` function (the principle described in Section 5.2.3). Above all, all the ETH and most of the CBC held by the destructed contract accounts are permanently locked in Ethereum.

Example: One smart contract named INSIGHTSNETWORKCONTRIBUTIONS (Address: 0x97eC9BFb0F6672C358620615a1E4dE0348Aea05c) is discovered by CLUE as destructed contract with locked cryptocurrencies. It has been transferred more than

Table 5.2: Top five transactions transferring ETH of one destructed smart contract account.

Rank	Transaction hash	Locked ETH	Value
1	0x4f64bd65c438e265be9ff7d8042018670578ba547bdbd49e3cf42d972eb118b4	50.00	18,654.50$
1	0xcd4902aa0a4e77074c6adebfc6a2455029341a270729429993a83d70af67582a	50.00	18,654.50$
3	0xa9bec2803452fd67ba7e09643d38560fb130af662cfa788db169a75a761b7bd6	27.00	10,073.43$
4	0x3403c6ec86697a836a1efb3e61e00c09ad914f021f0d05b6c1543ae1f59f9615	25.00	9,327.25$
5	0x9e4589470f92c14c62944dddba8f00d6b3143d7bddcc5608db5ef73ed4204b06	23.53	8,778.81$
Total	N/A	175.53	65,488.49$

208 ETH after its destruction (Transaction hash: 0x7e805cccba8ab2a1bac991ee0d0 33d6567864558e9e266c1087e607c63b2f0a9), which is worth more than 79 thousand dollars. Furthermore, all the CBC held by this destructed smart contract is also locked, which is worth about six dollars. Although there are many transactions sent to this destructed contract account, most of the locked ETH is related to several transactions. The top five transactions transferred ETH to this account are listed in Table 5.2. 84.4% (175.53ETH) of all its locked ETH is due to these top five transactions.

Attacked Parity contract

```
431  function() payable {
432    // just being sent some cash?
433    if (msg.value > 0)
434      Deposit(msg.sender, msg.value);
435    else if (msg.data.length > 0)
436      _walletLibrary.delegatecall(msg.data);
437  }
......
455  address constant _walletLibrary =
0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4;
```

Figure 5.14: Fallback function in the Solidity sources of Parity wallet contract.

In Nov. of 2017 [49], the attacker escalated his privilege and destructed the multi-sig library of Parity wallets, leading to all the ETH and most of the CBC held by wallet contracts that depend on the library locked permanently. The fallback function in the Solidity sources of Parity wallet contract is shown in Figure 5.14. One smart contract can only have one fallback function, which has no function name. In this fallback function, the wallet contract calls the library contract, whose address is hardcoded in Line 455. Many critical functions are written in the called wallet library, such as function `execute()` used for ETH transfer and function `kill()` used for contract account's destruction.

The attacker destructed the wallets' library in the following process. First, the attacker called the library's functions `initWallet()` and `initMultiowned()` through the fallback function, to escalate his/her privilege. Second, the attacker destructed the library contract through calling function `kill()`. After the library's destruction, all the wallet contracts can no longer call the library and executing its functions. Therefore, all the ETH stored in the attacked Parity wallet contracts is permanently locked. Furthermore, all the CBC held by the attacked wallet contracts is also locked. This is because the wallet contract cannot call the ERC20 contracts, whose details will be analyzed in Section 5.7.1.

Example: There are many attacked wallet contracts with much value of ETH, which are listed in Table 5.3. One attacked Parity wallet contract (Address: 0x0da3cB 3046F72fcbb49edF01B04AB6efc6C0D8DC) discovered by Clue stores 2,576.35ETH. After the attack, there were 17.88ETH transferred to this wallet contract. Furthermore, there is also 2.09$ value of CBC locked in it. If these wallet accounts with locked cryptocurrencies can be detected and alerted in time, the users might no longer transfer cryptocurrencies to it, which can help users to save money.

Table 5.3: Top ten attacked Parity wallet contracts with locked ETH detected by CLUE.

Rank	**Attacked Parity wallet**	**Locked ETH**	**Value**
1	0x3bfc20f0b9afcace800d73d2191166ff16540258	306,276.62	114,268,744.16$
2	0x376c3e5547c68bc26240d8dcc6729fff665a4448	114,939.00	42,882,591.51$
3	0x43ab622752d766d694c005acfb78b1fc60f35b69	21,704.33	8,097,668.48$
4	0xc7cd9d874f93f2409f39a95987b3e3c738313925	16,475.53	6,146,855.49$
5	0xdb0e7d784d6a7ca2cbda6ce26ac3b1bd348c06f8	6,925.00	2,583,648.25$
6	0x49eafa4c392819c009eccdc8d851b4e3c2dda7d0	4,524.98	1,688,224.79$
7	0xbe17d91c518f1743aa0556425421d59de0372766	4,360.67	1,626,922.37$
8	0x41849f3bd33ced4a21c73fddd4a595e22a3c2251	3,263.66	1,217,638.91$
9	0x8655d6bf4abd2aa47a7a4ac19807b26b7609b61d	3,000.00	1,119,270.00$
10	0x0da3cb3046f72fcbb49edf01b04ab6efc6c0d8dc	2,576.35	961,210.42$
Total	N/A	484,046.14	180,592,774.37$

5.6.2 Locked Cryptocurrencies in EOA

`0x0` account

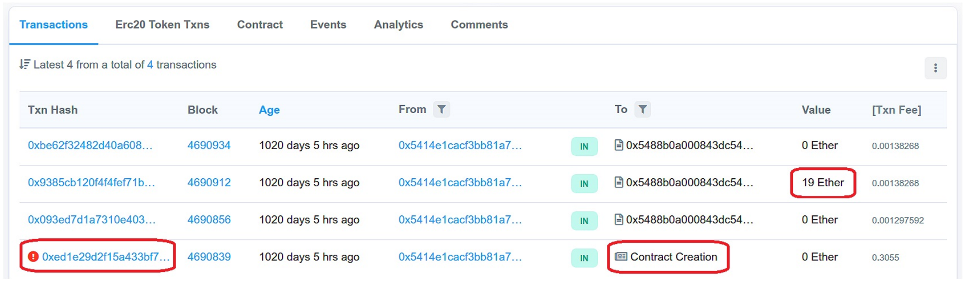
Because of users' unreasonable behaviors, their cryptocurrencies might be locked in the account whose address is `0x0`. Because the `0x0` account never stores any code, we classify it as EOA. Furthermore, no user has the private key of the `0x0` account and it cannot send out transaction. Therefore, all the ETH and CBC held by the `0x0` account are permanently locked. The cryptocurrencies are locked in the `0x0` account due to two scenarios: unreasonable mining and unreasonable cryptocurrency transfer, whose details will be analyzed in Section 5.7.2.

Example: The mining pool manager of Spark-Pool wrongly sent rewards to the `0x0` account in one transaction (Hash: 0x1920b021d6e3d637bbc72df4ea4f40032409fff b1dfa2de2e74cf009ffb08c06), leading to 5$ value of ETH permanently locked in the `0x0` account. For another example, one developer wrongly writes `withdraw(uint256 amount)` function's destination address as `0x0` and locks 20 ETH in one transaction

(Hash: 0xa6b5a31cbc29b6a54f5e046cf5af6e6ea1ecc184ae149d555f5e1cc153ecb0e1). Note that after the developer realized the bug, he/she destructed the contract (Address: 0x580E45f982a0A01cFab3B36B3Ec8Df63fcc5D290).

Contract-creation failure EOA

When the user deploys a smart contract in Ethereum, he/she will still receive one fake contract address if the contract-creation fails. Indeed, the received contract address does not exist in StateDB just after the contract-creation failure. However, some users might wrongly ignore the failure message and still transfer cryptocurrencies to the fake contract address, leading to cryptocurrencies locked permanently. Because the address with locked cryptocurrencies stores no code, we classify it as EOA.



Latest 4 from a total of 4 transactions

Txn Hash	Block	Age	From		To	Value	[Txn Fee]
0xbe62f32482d40a608...	4690934	1020 days 5 hrs ago	0x5414e1cacf3bb81a7...	IN	0x5488b0a000843dc54...	0 Ether	0.00138268
0x9385cb120f4f4fef71b...	4690912	1020 days 5 hrs ago	0x5414e1cacf3bb81a7...	IN	0x5488b0a000843dc54...	19 Ether	0.00138268
0x093ed7d1a7310e403...	4690856	1020 days 5 hrs ago	0x5414e1cacf3bb81a7...	IN	0x5488b0a000843dc54...	0 Ether	0.001297592
0xed1e29d2f15a433bf7...	4690839	1020 days 5 hrs ago	0x5414e1cacf3bb81a7...	IN	Contract Creation	0 Ether	0.3055

Figure 5.15: One contract-creation failure EOA with locked cryptocurrencies detected by CLUE.

Example: One EOA (Address: 0x5488b0a000843dc54b0e541dfb75c2927f92adc8) discovered by CLUE locks 19 ETH in value of 7,088.71 dollars and some CBC in value about seven dollars. After the user encountered an *out-of-gas* error during contract-creation, he still called the fake contract address three times, as shown in Figure 5.15.

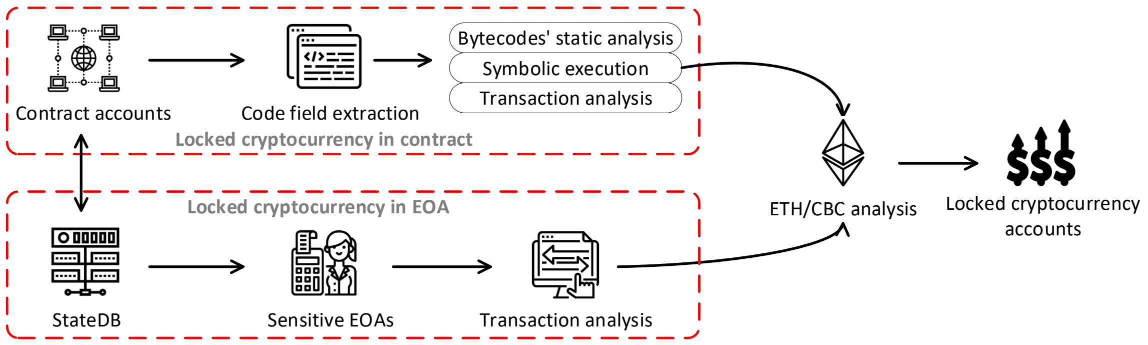


Figure 5.16: Overview of CLUE's architecture.

5.7 CLUE's Implementation

In this section, we detail the principle and implementation of CLUE. The overview of CLUE's architecture is shown in Figure 5.16, which mainly consists of two modules:

(1) Locked cryptocurrencies in contracts. In this module, CLUE detects two kinds of contract accounts with locked cryptocurrencies: destructed contract and attacked Parity contract. For destructed contract, we debug accounts' historical transactions and detect destructed contracts with locked cryptocurrencies through transaction trace analysis and ETH/CBC balance analysis. For attacked Parity contract, we statically analyze contracts' runtime bytecodes and detect wallet contracts with locked cryptocurrencies through symbolic execution and ETH/CBC balance analysis.

(2) Locked cryptocurrencies in EOAs. In this module, CLUE detects two kinds of EOAs with locked cryptocurrencies: `0x0` account and contract-creation failure account. For the `0x0` account, we analyze all its historical transactions to explain why cryptocurrencies are locked in the `0x0`. For contract-creation failure account, we export sensitive EOAs from StateDB and detect contract-creation failure accounts

with locked cryptocurrencies through transaction and ETH/CBC balance analysis.

Inside the two modules, we conduct plug-in development for the detection of each account category. Therefore, CLUE is scalable for more categories of accounts' detection in the future. Furthermore, we also plan to analyze other blockchain systems (e.g., Bitcoin, EOS), which will be described in Section 5.9.

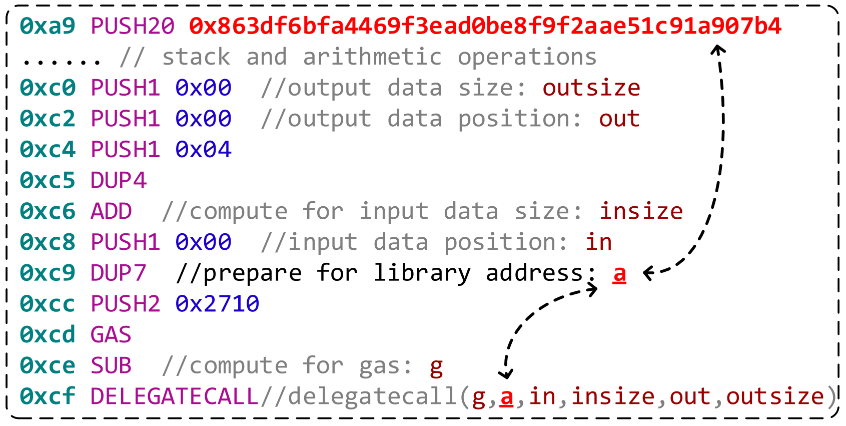
5.7.1 Detection of Locked Cryptocurrencies in Contract

Detection of destructed contracts

The detection of destructed contracts with locked cryptocurrencies is divided into four steps. First, for accounts stored in the StateDB, we debug their historical external transactions through Geth API `debug.traceTransaction()` [28]. From the execution trace of the external transaction, we analyze whether it ever executed the `SELFDESTRUCT` operation, which is used for destructing the contract account. Second, for the external transaction that executed `SELFDESTRUCT`, we leverage Ethereum RPC API to get the detailed information of the transaction. Because the execution of `SELFDESTRUCT` will produce internal transaction, we get the detailed information of the internal transaction according to the hash of external transaction. Third, leveraging the transaction's execution trace and detailed information, we analyze the specific address of the destructed contract account. If the type field of one internal transaction is "`suicide`", we can conclude that it is used for destructing the contract account. Then we export the sender address of the internal transaction, which is the address of destructed contract. Fourth, we analyze ETH/CBC balance of the destructed contract through Ethereum RPC-APIs [21, 25]. At last, the destructed contracts with locked cryptocurrencies can be discovered.

Detection of attacked Parity contracts

The detection of attacked Parity contracts with locked cryptocurrencies is divided into four steps. We use an example of one attacked Parity contract to describe the process, whose snippets of the runtime bytecodes are shown in Figure 5.17. First, for contract accounts stored in the StateDB, we export their runtime bytecodes from the code field $\sigma[a]_c$. Second, we statically analyze the bytecodes. Static analysis refers to analyzing the runtime bytecodes without attempting to run them [88]. In particular, we use DISASM [27] to disassemble the runtime bytecodes and detect hardcoded Parity library pattern. All the attacked parity wallet contracts hardcode the destructed library through pattern "`PUSH20 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`" (in Line 0xa9). Third, we leverage symbolic execution techniques to analyze the runtime bytecodes with the hardcode pattern. Symbolic execution refers to executing the codes with symbolic values, and we use OYENTE [134] as the symbolic execution engine. During the symbolic execution process of runtime bytecodes, we monitor the external call related operations (i.e., `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`). If we encounter external call operation's execution, we analyze its second operand P_a, which is used for the target address of the external call (in Line 0xc9 and 0xcf). If P_a is a real value and equals with the hardcoded Parity library's address, we can conclude that the corresponding analyzed contract account is an attacked Parity contract. Furthermore, the attacked Parity contract cannot call ERC20 contracts to transfer out its CBC. This is because P_a does not equal with ERC20 contracts' addresses or associated with transaction's input data. Fourth, we analyze the ETH/CBC balances for the detected contracts in the third step. At last, attacked Parity wallet contracts with locked cryptocurrencies can be discovered.



```
0xa9 PUSH20 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4
...... // stack and arithmetic operations
0xc0 PUSH1 0x00  //output data size: outsize
0xc2 PUSH1 0x00  //output data position: out
0xc4 PUSH1 0x04
0xc5 DUP4
0xc6 ADD  //compute for input data size: insize
0xc8 PUSH1 0x00  //input data position: in
0xc9 DUP7  //prepare for library address: a
0xcc PUSH2 0x2710
0xcd GAS
0xce SUB  //compute for gas: g
0xcf DELEGATECALL//delegatecall(g,a,in,insize,out,outsize)
```

Figure 5.17: Snippets of the disassembled runtime bytecodes of an attacked Parity wallet contract.

5.7.2 Detection of Locked Cryptocurrencies in EOA

Analysis of `0x0` account

We leverage transaction analysis to detect locked cryptocurrencies in the `0x0` account, demystifying reasons of the lock. As described in Section 5.6.2, the cryptocurrencies are locked in the `0x0` account due to two scenarios, i.e., unreasonable mining and unreasonable cryptocurrency transfer.

For unreasonable mining, the mining reward (i.e., ETH) might be locked in `0x0` account during pool mining or solo mining. If the miner does not correctly set his address (e.g., set as null) in the mining pool, the pool manager might send the miner's reward to the `0x0` account. We filter out sender addresses of all the `0x0`'s 26,243 historical external transactions (i.e., sent from EOAs) and discovered ten transactions with ETH sent from pool managers [36]. Similarly, if the miner does not correctly set his address during solo mining, Ethereum system might send rewards to the `0x0` account. Etherscan has listed all the 483.89 ETH wrongly sent to `0x0` during solo mining [1].

For unreasonable cryptocurrency transfer, the cryptocurrencies might be locked due to transaction setting errors or contract programming bugs. If the user does not correctly set transaction's destination address, the transaction might be sent to `0x0` by some clients. For example, one user forgot to set the destination address, and wrongly sent his Golem (one kind of CBC) to `0x0` [23]. We analyze all `0x0`'s historical external transactions. If one transaction's sender address is not a mining pool manager and its value field is not zero, we can conclude that it is a setting error transaction with locked ETH. Besides, if there exist bug(s) in contract's cryptocurrency transfer function, the contract might wrongly transfer cryptocurrencies to the `0x0`. Correct transfer functions should check the destination address to avoid cryptocurrencies being locked. We analyze all the 109,729 internal transactions (i.e., sent from smart contracts) sent to `0x0`. If one transaction's value field is not zero, we can conclude that its ETH is locked due to smart contract's programming bugs. Furthermore, all the CBC held by the `0x0` account is locked due to unreasonable transfers.

Detection of contract-creation failure EOAs

We leverage account state analysis and transaction analysis to detect contract-creation failure EOAs with locked cryptocurrencies, which is divided into three steps. First, we traverse the StateDB and filter out sensitive EOAs. The sensitive EOAs have the following state features: nonce $\sigma[a]_n$ is zero, and code $\sigma[a]_c$ is empty. The sensitive EOAs with these features never send out any transaction. $\sigma[a]_c$ field is empty indicates that the account a is an EOA. For an EOA, its $\sigma[a]_n$ field stores the number of transactions sent from it. Second, leveraging Ethereum RPC-API, we fetch and analyze sensitive EOA's oldest transaction, to verify that it encountered an error and returned a smart contract address. As described in Section 5.6.2, the contract-creation transaction will also return a fake contract address when it fails with errors. Third, we analyze ETH/CBC balance of the detected EOAs in the second step through Ethereum RPC-APIs. At last, contract-creation failure accounts with locked cryptocurrencies can be discovered.

5.8 CLUE's Evaluation

In this section, we carry out experiments to answer the following research questions:

RQ1 (Quantity): How much value of locked cryptocurrencies can be detected by CLUE?

RQ2 (Accuracy): To what extent can CLUE accurately discover locked cryptocurrencies?

5.8.1 RQ1: Quantity

We evaluate the quantity of locked cryptocurrencies detected by CLUE, whose statistics are shown in Table 5.4. Note that all cryptocurrency values are measured in US dollars. (❍: discovered candidate accounts before ETH/CBC balance analysis. ●: accounts with locked cryptocurrencies. ♣: external transactions. ♠: internal transactions.)

Applying CLUE to all Ethereum StateDB data, we totally discover 1,091,796,292.09\$ value of locked cryptocurrencies. The related accounts' addresses for each category and analyzed transaction data are published on https://figshare.com/s/2f2d

97899f5ecc361b21. For the destructed contracts, many of them were created due to DoS attacks in 2016 [81]. The attacker created large amount of smart contracts and destructed them through SELFDESTRUCT operation. Most of these destructed contracts are not called any more by normal users. Therefore, most of the destructed contracts do not lock any cryptocurrency. For the attacked Parity contracts, we totally discover 658 related accounts, while Etherscan only tags 153 of them [44]. For the 0x0 account, there is no locked CBC due to unreasonable mining, because Ethereum only rewards the miner with ETH. For contract-creation failure EOAs, their locked cryptocurrencies' value is small, because users might stop calling these accounts after they realize the contract-creation failure. The locked CBC of destructed contracts does not be transferred out during contracts' destruction, which leads to more locked CBC than ETH. Furthermore, all these detected accounts might lock more cryptocurrencies with Ethereum's running, and we also plan to measure locked cryptocurrencies' time accumulation in our future work.

Answer to RQ1 (Quantity): For the proposed four kinds of Ethereum accounts, we totally discover 1,091,796,292.09$ value of cryptocurrencies locked in them.

Table 5.4: Statistics of locked cryptocurrencies and accounts detected through CLUE.

Category	Account type	Account with locked cryptocurrencies	Discovered account	Transaction (●)	Locked ETH	Locked CBC	Total value
❶	Smart contract	Destructed contract	5,916,076❍ | 173●	138,706♣ | 55,533♠	123,841.02$	25,036,305.09$	25,160,146.11$
❷	Smart contract	Attacked Parity contract	658❍ | 203●	8,348♣ | 6,599♠	190,060,328.19$	950,380.79$	191,010,708.98$
❸	EOA	0x0: Unreasonable mining	N/A (1)	26,243♣ | 109,729♠	180,867.55$	N/A (0$)	180,867.55$
		0x0: Unreasonable transfer			2,723,595.04$	872,705,278.36$	875,428,873.40$
❹	EOA	Contract-creation failure EOA	3,720❍ | 191●	490♣ | 6♠	15,640.76$	55.27$	15,696.03$
Total	N/A	N/A	5,920,455❍ | 568●	173,787♣ | 171,867♠	193,104,272.58$	898,692,019.51$	1,091,796,292.09$

5.8.2 RQ2: Accuracy

We evaluate the accuracy of locked cryptocurrencies detected by CLUE through Etherscan. Some of the accuracy evaluation need manual analysis. To avoid the threat of inter-rater reliability, we ask three different people to evaluate their accuracy.

For destructed contracts, we check all the 173 discovered accounts through Etherscan. All of them have been tagged *"Self-Destruct"*, and they all have more than zero value of ETH/CBC. Furthermore, there is no ETH/CBC transferred out after their destruction. Similarly, all the 203 attacked Parity wallets have more than zero value of ETH/CBC, and their ETH/CBC never be transferred out after the Parity attack (Transaction hash: 0x47f7cff7a5e671884629c93b368cb18f58a993f4b19c2a53a8662e3f 1482f690). In addition, we decompile these contracts leveraging PANORAMIX [42], and they all call the attacked Parity wallets' library. Note that although Etherscan displays that some CBC is transferred out from `0x0`, it is a developer's programming error [44], because Etherscan only monitors CBC's transfer through ERC20 contracts' events (Line 9 in Figure 5.2). Indeed, because nobody has the private key of `0x0`, its ETH/CBC is locked permanently. For the contract-creation failure accounts, we check all the 191 discovered accounts that lock cryptocurrencies through Etherscan. All of these accounts encountered errors during contract-creation, and they all have more than zero value of ETH/CBC. Also, their ETH/CBC is never transferred out.

Answer to RQ2 (Accuracy): 100% of the 568 accounts discovered by CLUE store cryptocurrencies, and all of these cryptocurrencies are locked permanently.

5.9 Discussion

In this section, we discuss some limitations, the corresponding solutions and future work.

(1) We will extend Gasper by identifying more gas-inefficient patterns and the corresponding efficient patterns. Moreover, we plan to cover all these patterns and improve compilers to produce gas-efficient bytecodes.

(2) We propose and detect four categories of accounts with locked cryptocurrencies in Ethereum, while there might also exist other categories. In our future work, we plan to analyze more categories of accounts with locked cryptocurrencies.

(3) We run Clue on all the Ethereum accounts' data. Although the number of discovered accounts with locked cryptocurrencies is small (i.e., 568), the value of locked cryptocurrencies is great. To the best of our knowledge, there is still no research of how many accounts with locked cryptocurrencies exist in Ethereum, and our work fills this gap.

(4) In other blockchain systems (e.g., Bitcoin, EOS), there might also exist locked cryptocurrencies. The Clue version in this chapter cannot be directly applied to other blockchain systems, because of different system architectures. However, we plan to improve Clue to detect more cryptocurrencies in other blockchain systems in future work.

(5) All the CBC analyzed in this chapter are compliant with the ERC20 standard, and we will analyze more comprehensive standard CBC (e.g., ERC721) in our future work. After some improvements, we will also open-source Clue.

(6) For symbolic execution techniques, path explosion and timeout exception are

common threats. To avoid these threats, before symbolic execution, we statically analyze the runtime bytecodes to detect contracts with hardcode pattern (described in Section 5.7.1). The combination of static analysis and symbolic execution improves CLUE's performance and reduces exceptions.

(7) As Ethereum system is running, there will be more locked cryptocurrencies, and we plan to analyze its time accumulation in our future work. Although the cryptocurrencies locked in Ethereum cannot be used or transferred out by users, CLUE still can help users to save money. Because CLUE can remind users not to call the discovered accounts to avoid locking more cryptocurrencies. Furthermore, CLUE can also reveal why cryptocurrencies are locked, which can remind users to avoid unreasonable behaviors.

5.10 Brief Summary

We perform the *first* investigation to expose that lots of smart contracts, generated by the recommended compiler Solidity, contain gas-inefficient bytecodes, which can be replaced with gas-efficient bytecodes to save money. In particular, we identify seven gas-inefficient patterns belonging to two categories. Moreover, we propose and develop GASPER that leverages symbolic execution to automatically discover three representative gas-inefficient patterns in bytecode. By applying GASPER to all deployed smart contracts, we find that 93.5%, 90.1% and 80% smart contracts suffer from these three patterns, respectively.

In this chapter, we also analyze cryptocurrencies locked permanently in Ethereum. We define four categories of accounts with locked cryptocurrencies and implement a tool named CLUE, which discovers more than one billion dollars value of locked cryptocurrencies. We also analyze why these cryptocurrencies are locked, which can help users/developers to avoid losing money. We will analyze locked cryptocurrencies in more account types and other blockchain systems (e.g., Bitcoin, EOS) in our future work.

Chapter 6

Conclusion

Since its inception, blockchain technology has shown promising application prospects from cryptocurrency to a variety of forms, such as medicine, economics, cloud computing, and so on. As the most popular blockchain system that supports smart contract, Ethereum can complete one million transactions per day. Since blockchain is one of the core technology in FinTech industry, users are very concerned about its security. Some security vulnerabilities and attacks have been recently reported. In Chapter 2, we systematically survey security issues for blockchain systems.

More than eight million smart contracts have already been deployed in Ethereum, while only less than 1% are open-source. Unfortunately, facing the bytecodes of deployed smart contracts, it is difficult to quickly and comprehensively understand their details. In Chapter 3, we describe the runtime bytecodes of smart contracts in natural language.

Ethereum has two kinds of accounts: EOA and contract account. However, not all accounts should be kept. We regard the worthless accounts that deserve to be removed without affecting the normal operations of users and other accounts as erasable accounts. Erasable accounts not only waste system resources and affect the efficiency of blockchain, but also easily waste users' money. We characterize erasable accounts of Ethereum in Chapter 4.

Gas is the execution fee for running smart contracts in Ethereum. However, we find that under-optimized smart contracts cost more gas than necessary, and therefore the miners or users will be overcharged. There are already more than 296 thousand kinds of cryptocurrencies built on Ethereum. However, not all cryptocurrencies can be controlled by users. We analyze under-optimized smart contracts and locked cryptocurrencies in Chapter 5.

6.1 Summary of Contribution

We make the following contributions in this thesis:

1. We conduct systematic examination on the security of blockchain systems. We survey real attacks and analyze the exploited vulnerabilities. Furthermore, we summarize practical academic achievements for enhancing the security of blockchain.

2. We propose and implement a system named Stan, which can analyze the runtime bytecodes of smart contract and automatically describe its interfaces in natural language, enabling users to quickly and thoroughly understand closed-source contracts.

3. We design and implement a tool named Glaser to discover erasable accounts by analyzing the StateDB of Ethereum. It leverages program analysis techniques

to discover contract accounts with worthless runtime bytecodes, and employs state field and transaction analysis to discover EOAs that no one owns their private keys.

4. We develop a tool named Gasper for discovering gas-inefficient patterns in bytecodes. Gasper leverages symbolic execution and it can locate three representative patterns. We find that more than 80% smart contracts suffer from these three patterns, respectively.

5. We conduct the systematic investigation on locked cryptocurrencies in Ethereum. We define four categories of accounts with locked cryptocurrencies and develop a tool named Clue to discover them. Results show that there are more than one billion dollars value of cryptocurrencies locked in Ethereum.

6.2 Future Work

We plan to conduct the following improvements in our future work.

For Chapter 3, we plan to improve cloud instance's configurations, and use more significant timeout threshold to reduce the number of timeout cases. Furthermore, we consider improving Stan's performance with faster static analysis techniques and evaluating Stan with more comprehensive bytecodes datasets. We will also build more and better syntax trees, and add more common word abbreviations in Ethereum to Stanford parser's rule libraries to improve SWUM analysis. At last, we will conduct more features' and behavior' analysis to improve Stan's functionalities.

For Chapter 4, Glaser can not only discover erasable accounts that already exist in Ethereum, but also erasable accounts that might be created in future. Some kinds of accounts analyzed by GLASER might also be created in future, and GLASER might discover more erasable accounts. As GLASER focuses on five kinds of erasable accounts in Ethereum, we will detect more kinds of erasable accounts in future. We will also analyze erasable accounts in other blockchain systems.

For Chapter 5, we plan to identify more gas-inefficient patterns and extend GASPER to cover all these patterns. We also plan to improve compilers to produce gas-efficient bytecode. Furthermore, we plan to analyze more categories of accounts with locked cryptocurrencies in Ethereum, and improve CLUE to detect more cryptocurrencies in other blockchain systems. At last, we will analyze more comprehensive standard CBC (e.g., ERC721) in our future work.

6.3 Insights

Based on the above research, we list a few future directions to stir up research efforts into this area.

First, nowadays the most popular consensus mechanism used in blockchain is PoW. However, a major disadvantage of PoW is the waste of computing resources. To solve this problem, Ethereum is trying to develop a hybrid consensus mechanism of PoW and PoS. Conducting researches and developing more efficient consensus mechanisms will make a significant contribution to the development of blockchain. Second, with the growth of the number of feature-rich DApps, the privacy leakage risk of blockchain will be more serious. A DApp itself, as well as the process of communication between the DApp and Internet, are both faced with privacy leakage risks. There are some interesting techniques that can be applied in this problem: code obfuscation, application hardening, execution trusted computing (e.g., Intel SGX), etc. Third, the blockchain will produce a lot of data, including block information, transaction data, contract bytecodes, etc. However, not all of the data stored in blockchain is valid and useful. For example, many smart contracts are never be executed after their deployments. An efficient data cleanup and detection mechanism is desired to improve the execution efficiency of blockchain systems.

References

[1] 0x0 mined blocks. https://etherscan.io/address/0x0000000000000000000000000000000000000000#mine Accessed June, 2021.

[2] 51% attack. http://cryptorials.io/glossary/51-attack Accessed June, 2021.

[3] Behind the biggest bitcoin heist in history: Inside the implosion of mt.gox. http://www.thedailybeast.com/articles/2016/05/19/behind-the-biggest-bitcoin-heist-in-history-inside-the-implosion-of-mt-gox.html Accessed June, 2021.

[4] Best-performing commodity is bitcoin? http://www.marketwatch.com/story/and-2016s-best-performing-commodity-is-bitcoin-2016-12-22 Accessed June, 2021.

[5] BGP hijacking for cryptocurrency profit. https://www.secureworks.com/research/bgp-hijacking-for-cryptocurrency-profit Accessed June, 2021.

[6] Bitcoin miners ditch ghash.io pool over fears of 51% attack. http://www.coindesk.com/bitcoin-miners-ditch-ghash-io-pool-51-attack Accessed June, 2021.

[7] Bitcoin wallet. https://blockchain.info/wallet/# Accessed June, 2021.

[8] Blockflow. https://blockflow.net Accessed June, 2021.

[9] Buy ethereum: Eth/btc, eth/usd and eth/eur. https://cex.io/buy-ethereum Accessed June, 2021.

[10] Contracts with verified source codes. https://etherscan.io/contractsVerified Accessed June, 2021.

[11] Critical update re: Dao vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability Accessed June, 2021.

[12] Cryptocurrency market capitalizations. https://coinmarketcap.com Accessed June, 2021.

[13] The current state of ransomware: Ctb-locker. https://news.sophos.com/en-us/2015/12/31/the-current-state-of-ransomware-ctb-locker Accessed June, 2021.

[14] Darkwallet. https://darkwallet.is Accessed June, 2021.

[15] Eip-158: State clearing. https://eips.ethereum.org/EIPS/eip-158 Accessed June, 2021.

[16] Erc20 tokens. https://etherscan.io/tokens Accessed June, 2021.

[17] Ethereum function signature database. https://www.4byte.directory Accessed June, 2021.

[18] Ethereum improvement proposals. https://eips.ethereum.org/erc Accessed June, 2021.

[19] Ethereum natural specification format. https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format Accessed June, 2021.

[20] Ethereum network comes across yet another dos attack. http://www.newsbtc.com/2016/09/23/ethereum-dao-attack-attack-platforms-credibility Accessed June, 2021.

[21] Ethereum rpc-api. https://eth.wiki/json-rpc/API Accessed June, 2021.

[22] Ethereum transaction chart. https://etherscan.io/chart/tx Accessed June, 2021.

[23] Ethereum wallet sends to 0x0. https://www.reddit.com/r/ethereum/comments/5rd4co/ethereum_wallet_sends_to Accessed June, 2021.

[24] Etherscan. https://etherscan.io Accessed June, 2021.

[25] Etherscan: Developer apis. https://etherscan.io/apis Accessed June, 2021.

[26] Etherscan: Gasprice history. https://etherscan.io/charts/gasprice Accessed June, 2021.

[27] Evm disassembler. https://github.com/Arachnid/evmdis Accessed June, 2021.

[28] Go ethereum: Official go implementation of the ethereum protocol. https://geth.ethereum.org/docs/rpc/ns-debug#debug_tracetransaction Accessed June, 2021.

[29] It's official: Bitcoin was the top performing currency of 2015. http://money.visualcapitalist.com/its-official-bitcoin-was-the-top-performing-currency-of-2015/ Accessed June, 2021.

[30] Jeb. https://www.pnfsoftware.com/jeb/evm Accessed June, 2021.

[31] Know your ransomware: Ctb-locker. https://www.secalliance.com/blog/ransomware-ctb-locker Accessed June, 2021.

[32] Kovan - stable ethereum public testnet. https://github.com/kovan-testnet/proposal Accessed June, 2021.

[33] Language-check. https://github.com/myint/language-check Accessed June, 2021.

[34] Long-term gas cost changes for io-heavy operations to mitigate transaction spam attacks. https://github.com/ethereum/EIPs/issues/150 Accessed June, 2021.

[35] Manticore. https://github.com/trailofbits/manticore Accessed June, 2021.

[36] Mining pools. https://etherscan.io/accounts/label/mining Accessed June, 2021.

[37] Mythril. https://github.com/ConsenSys/mythril Accessed June, 2021.

[38] Number of daily bitcoin transactions worldwide. https://www.statista.com/statistics/730806/daily-number-of-bitcoin-transactions Accessed June, 2021.

[39] Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum Accessed June, 2021.

[40] Oyente. https://github.com/melonproject/oyente Accessed June, 2021.

[41] Pakistan hijacks youtube. http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1 Accessed June, 2021.

[42] Panoramix. https://github.com/eveem-org/panoramix Accessed June, 2021.

[43] Parity. https://parity.io Accessed June, 2021.

[44] Parity bug. https://etherscan.io/accounts/label/parity-bug Accessed June, 2021.

[45] Parity multi-sig library self-destruct. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct Accessed June, 2021.

[46] Proof of authority chains. https://github.com/paritytech/parity/wiki/Proof-of-Authority-Chains Accessed June, 2021.

[47] Proof of elapsed time (poet). http://intelledger.github.io Accessed June, 2021.

[48] R3. https://www.r3.com Accessed June, 2021.

[49] Security alert: Parity wallet (multi-sig wallets). https://www.parity.io/security-alert-2 Accessed June, 2021.

[50] Smart contracts: The blockchain technology that will replace lawyers. https://blockgeeks.com/guides/smart-contracts Accessed June, 2021.

[51] Solidity. https://solidity.readthedocs.io Accessed June, 2021.

[52] Surveymonkey. https://www.surveymonkey.com Accessed June, 2021.

[53] Tether usd. https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7 Accessed June, 2021.

[54] Token tracker. https://etherscan.io/tokens Accessed June, 2021.

[55] A torpath to torcoin, proof-of-bandwidth altcoins for compensating relays. https://www.smithandcrown.com/open-research/a-torpath-to-torcoin-proof-of-bandwidth-altcoins-for-compensating-relays Accessed June, 2021.

[56] Total ether supply and market capitalization. https://etherscan.io/stat/supply Accessed June, 2021.

[57] Town crier. http://www.town-crier.org Accessed June, 2021.

[58] Uk national risk assessment of money laundering and terrorist financing. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/468210/UK_NRA_October_2015_final_web.pdf Accessed June, 2021.

[59] Wannacry ransomware attack. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack Accessed June, 2021.

[60] What is cryptocurrency: Everything you need to know. https://blockgeeks.com/guides/what-is-cryptocurrency Accessed June, 2021.

[61] Wordninja. https://github.com/keredson/wordninja Accessed June, 2021.

[62] The "yellow paper": Ethereum's formal specification. https://github.com/ethereum/yellowpaper Accessed June, 2021.

[63] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *Preprint arXiv:1708.03778*, 2017.

[64] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: gas analysis and optimization for ethereum smart contracts. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2020.

[65] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*, 2018.

[66] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. Running on fumes. In *Proceedings of the International Conference on Verification and Evaluation of Computer and Communication Systems*, 2019.

[67] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *Proceedings of the 7th SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

[68] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[69] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Proceedings of the International Conference on Principles of Security and Trust*, 2017.

[70] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *Proceedings of the International Conference on Open and Big Data*, 2016.

[71] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *Proceedings of the 1st Workshop on Trusted Smart Contracts*, 2017.

[72] Alex Biryukov and Sergei Tikhomirov. Deanonymization and linkability of cryptocurrency transactions based on network analysis. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2019.

[73] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[74] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. In *Preprint arXiv:1809.03981*, 2018.

[75] Eli Bressert. *SciPy and NumPy: an overview for developers.* O'Reilly Media Inc, 2012.

[76] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, 1997.

[77] Paweł Bylica, Łukasz Gleń, Piotr Janiuk, Aleksandra Skrzypczak, and Artur Zawłocki. A probabilistic nanopayment scheme for golem. 2015.

[78] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. scompile: Critical path identification and analysis for smart contracts. In *Proceedings of the International Conference on Formal Engineering Methods*, 2019.

[79] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. In *IEEE Transactions on Emerging Topics in Computing*, 2020.

[80] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[81] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to

defend against under-priced dos attacks. In *Proceedings of the International Conference on Information Security Practice and Experience*, 2017.

[82] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. Towards saving money in using smart contracts. In *Proceedings of the 40th IEEE International Conference on Software Engineering*, 2018.

[83] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[84] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. Understanding ethereum via graph analysis. In *Proceedings of the IEEE Conference on Computer Communications*, 2018.

[85] Weili Chen, Tuo Zhang, Zhiguang Chen, Zibin Zheng, and Yutong Lu. Traveling the token world: A graph analysis of ethereum erc20 token ecosystem. In *Proceedings of the World Wide Web Conference*, 2020.

[86] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *Proceedings of the World Wide Web Conference*, 2018.

[87] Zhen Cheng, Xinrui Hou, Runhuai Li, Yajin Zhou, Xiapu Luo, Jinku Li, and Kui Ren. Towards a first step to understand the cryptocurrency stealing attack on ethereum. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[88] Brian Chess and Gary McGraw. Static analysis for security. In *IEEE S&P*, 2004.

[89] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. O'Reilly Media Inc, 2013.

[90] Nicolas Christin. Traveling the silk road: A measurement analysis of a large anonymous online marketplace. In *Proceedings of the 22nd international conference on World Wide Web*, 2013.

[91] Jacob Stenum Czepluch, Nikolaj Zangenberg Lollike, and Simon Oliver Malone. The use of block chain technology in different application domains. In *The IT University of Copenhagen*, 2015.

[92] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[93] Peter D DeVries. An analysis of cryptocurrency, bitcoin, and the future. In *International Journal of Business Management and Commerce*, 2016.

[94] Monika Di Angelo and Gernot Salzer. Collateral use of deployment code for smart contracts in ethereum. In *Proceedings of the 10th IFIP International Conference on New Technologies, Mobility and Security*, 2019.

[95] Monika Di Angelo and Gernot Salzer. Tokens, types, and standards: Identification and utilization in ethereum. In *Proceedings of the International Conference of Decentralized Applications and Infrastructures*, 2020.

[96] Ali Dorri, Salil S Kanhere, Raja Jurdak, and Praveen Gauravaram. Blockchain for iot security and privacy: The case study of a smart home. In *Proceedings of the IEEE Percom Workshop on Security Privacy and Trust in the Internet of Thing*, 2017.

[97] Ariel Ekblaw, Asaph Azaria, John D Halamka, and Andrew Lippman. A case study for blockchain in healthcare:“medrec” prototype for electronic health records and medical research data. https://www.media.mit.edu/publications/medrec-whitepaper Accessed June, 2021.

[98] Christian Esposito, Alfredo De Santis, Genny Tortora, Henry Chang, and Kim-Kwang Raymond Choo. Blockchain: A panacea for healthcare cloud-based data security and privacy? In *IEEE Cloud Computing*, 2018.

[99] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Proceedings of the 18th International Conference Financial Cryptography and Data Security*, 2014.

[100] Josselin Feist, Gustavo Grieco, and Alex Groce. journal: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019.

[101] Gianni Fenu, Lodovica Marchesi, Michele Marchesi, and Roberto Tonelli. The ico phenomenon and its relationships with ethereum smart contract environment. In *Proceedings of the International Workshop on Blockchain Oriented Software Engineering)*, 2018.

[102] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[103] Michael Fröwis and Rainer Böhme. In code we trust? In *Proceedings of the Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2017.

[104] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. Detecting token systems on ethereum. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2019.

[105] Albert Gatt and Ehud Reiter. Simplenlg: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, 2009.

[106] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[107] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *Proceedings of the IEEE International Conference on Software Engineering*, 2019.

[108] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proceedings of the ACM on Programming Languages*, 2018.

[109] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Proceedings of the International Conference on Principles of Security and Trust*, 2018.

[110] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. In *Proceedings of the ACM on Programming Languages*, 2017.

[111] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[112] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium*, 2018.

[113] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2017.

[114] Sabrina T Howell, Marina Niessner, and David Yermack. Initial coin offerings: Financing growth with cryptocurrency token sales. In *The Review of Financial Studies*, 2020.

[115] Xinwen Hu, Yi Zhuang, Shang-Wei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. A security type verifier for smart contracts. In *Computers & Security*, 2021.

[116] Steve Huckle, Rituparna Bhattacharya, Martin White, and Natalia Beloff. Internet of things, blockchain and shared economy applications. In *Procedia Computer Science*, 2016.

[117] Petter Hurich. The virtual is real: An argument for characterizing bitcoins as private property. In *Banking & Finance Law Review*, 2016.

[118] Ru Ji, Ningyu He, Lei Wu, Haoyu Wang, Guangdong Bai, and Yao Guo. Deposafe: Demystifying the fake deposit vulnerability in ethereum smart contracts. In *Preprint arXiv:2006.06419*, 2020.

[119] Ari Juels, Ahmed Kosba, and Elaine Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[120] Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. Blocksci: Design and applications of a blockchain analysis platform. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[121] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[122] Ghassan O Karame, Elli Androulaki, Marc Roeschlin, Arthur Gervais, and Srdjan Čapkun. Misbehavior in bitcoin: A study of double-spending and accountability. In *ACM Transactions on Information and System Security*, 2015.

[123] Karl. Ethereum eclipse attacks. http://e-collection.library.ethz.ch/view/eth:49728 Accessed June, 2021.

[124] Karl. *Security of Blockchain Technologies.* PhD thesis, Swiss Federal Institute of Technology, 2016.

[125] Aggelos Kiayias and Giorgos Panagiotakos. On trees, chains and fast transactions in the blockchain. In *IACR ePrint Archive*, 2016.

[126] Lucianna Kiffer, Dave Levin, and Alan Mislove. Analyzing ethereum's contract topology. In *Proceedings of the Internet Measurement Conference*, 2018.

[127] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[128] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[129] Nitesh Kumar, Ajay Singh, Anand Handa, and Sandeep Kumar Shukla. Detecting malicious accounts on the ethereum blockchain with supervised learning. In *Proceedings of the International Symposium on Cyber Security Cryptography and Machine Learning*, 2020.

[130] Xiaoqi Li, Ting Chen, Xiapu Luo, and Jiangshan Yu. Characterizing erasable accounts in ethereum. In *Proceedings of the 23rd Information Security Conference*, 2020.

[131] Xiaoqi Li, Ting Chen, Xiapu Luo, Tao Zhang, Le Yu, and Zhou Xu. Stan: Towards describing bytecodes of smart contract. In *Proceedings of the 20th IEEE International Conference on Software Quality, Reliability and Security*, 2020.

[132] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. In *Future Generation Computer Systems*, 2017.

[133] Xueping Liang, Sachin Shetty, Deepak Tosh, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2017.

[134] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[135] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[136] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smart pool: Practical decentralized pooled mining. In *Proceedings of the USENIX Security Symposium*, 2017.

[137] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. In *preprint arXiv:1910.02945*, 2019.

[138] Matteo Marescotti, Martin Blicha, Antti EJ Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing exact worst-case gas consumption for smart contracts. In *International Symposium on Leveraging Applications of Formal Methods*, 2018.

[139] Hartwig Mayer. Ecdsa security in bitcoin and ethereum: a research survey. http://blog.coinfabrik.com/wp-content/uploads/2016/06/ECDSA-Security-in-Bitcoin-and-Ethereum-a-Research-Survey.pdf Accessed June, 2021.

[140] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into text. In *Proceedings of the conference on empirical methods in natural language processing*, 2004.

[141] Andrew Miller, Malte Möser, Kevin Lee, and Arvind Narayanan. An empirical analysis of linkability in the monero blockchain. In *Preprint arXiv:1704.04299*, 2017.

[142] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In *Proceedings of the 9th Annual HITB Security Conference*, 2018.

[143] Christopher Natoli and Vincent Gramoli. The balance attack against proof-of-work blockchains: The r3 testbed as an example. In *Preprint arXiv:1612.09426*, 2016.

[144] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[145] Joakim Nivre, Marie-Catherine De Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the 10th International Conference on Language Resources and Evaluation*, 2016.

[146] Erik Nordström. Personal clouds: Concedo. Master's thesis, Lulea University of Technology, 2015.

[147] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

[148] Purathani Praitheeshan, Yi Wei Xin, Lei Pan, and Robin Doss. Attainable hacks on keystore files in ethereum wallets—a systematic analysis. In *Proceedings of the International Conference on Future Network Systems and Security*, 2019.

[149] Mayra Samaniego and Ralph Deters. Blockchain as a service for iot. In *Proceedings of the IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*, 2016.

[150] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, 2018.

[151] Atul Singh, Tsuen-Wan Ngan, Peter Druschel, and Dan S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of the 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies*, 2006.

[152] Siamak Solat and Maria Potop-Butucaru. Zeroblock: Preventing selfish mining in bitcoin. In *Preprint arXiv: 1605.02435*, 2016.

[153] Shahar Somin, Goren Gordon, and Yaniv Altshuler. Network analysis of erc20 tokens trading on ethereum blockchain. In *Proceedings of the International Conference on Complex Systems*, 2018.

[154] Shahar Somin, Goren Gordon, and Yaniv Altshuler. Social signals in the ethereum trading network. In *preprint arXiv:1805.12097*, 2018.

[155] Shahar Somin, Goren Gordon, Alex Pentland, Erez Shmueli, and Yaniv Altshuler. Erc20 transactions over ethereum blockchain: Network analysis and predictions. In *Preprint arXiv:2004.08201*, 2020.

[156] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[157] Matt Suiche. Porosity: A decompiler for blockchain-based smart contracts bytecode. In *Proceedings of the Defcon*, 2017.

[158] Jianjun Sun, Jiaqi Yan, and Kem ZK Zhang. Blockchain-based sharing services: What blockchain technology can contribute to smart cities. In *Financial Innovation*, 2016.

[159] Melanie Swan. *Blockchain: Blueprint for a new economy.* O'Reilly Media, 2015.

[160] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st IEEE International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[161] Florian Tramèr, Dan Boneh, and Kenny Paterson. Remote side-channel attacks on anonymous transactions. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[162] Petar Tsankov. Security analysis of smart contracts in datalog. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, 2018.

[163] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[164] Friedhelm Victor. Address clustering heuristics for ethereum. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2020.

[165] Friedhelm Victor and Bianca Katharina Lüders. Measuring ethereum-based erc20 token networks. In *Proceedings of the International Conference on Financial Cryptography and Data Security*, 2019.

[166] Zhiyuan Wan, Xin Xia, and Ahmed E Hassan. What is discussed about blockchain? a case study on the use of balanced lda and the reference architecture of a domain to capture online discussions about blockchain platforms across the stack exchange communities. In *IEEE Transactions on Software Engineering*, 2019.

[167] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. In *Proceedings of the VLDB Endowment*, 2018.

[168] Xiwei Xu, Cesare Pautasso, Liming Zhu, Vincent Gramoli, Alexander Ponomarev, An Binh Tran, and Shiping Chen. The blockchain as a software connector. In *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture*, 2016.

[169] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey. BGPmon: A real-time, scalable, extensible monitoring system. In *Proceedings of the Cybersecurity Applications Technology Conference for Homeland Security*, 2009.

[170] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing ethereum's gas mechanism. In *Proceedings of the IEEE European Symposium on Security and Privacy Workshops*, 2019.

[171] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. A multi-modal transformer-based code summarization approach for smart contracts. In *preprint arXiv:2103.07164*, 2021.

[172] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. In *Journal of Medical Systems*, 2016.

[173] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[174] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William Knottenbelt. Xclaim: Trustless, interoperable, cryptocurrency-backed assets. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[175] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *Proceedings of the IEEE 35th international conference on data engineering*, 2019.

[176] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[177] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *Proceedings of the IEEE International Conference on Software Engineering*, 2020.

[178] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. Txspector: Uncovering attacks in ethereum from transactions. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[179] Yu Zhang and Jiangtao Wen. The iot electric business model: Using blockchain technology for the internet of things. In *Peer-to-Peer Networking and Applications*, 2016.

[180] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. In *International Journal of Web and Grid Services*, 2018.

[181] Shunfan Zhou, Malte Möser, Zhemin Yang, Ben Adida, Thorsten Holz, Jie Xiang, Steven Goldfeder, Yinzhi Cao, Martin Plattner, Xiaojun Qin, et al. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[182] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: reverse engineering ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[183] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. In *IEEE Transactions on Software Engineering*, 2019.